**VŠB – Technická univerzita Ostrava**
**Fakulta elektrotechniky a informatiky**
**Katedra informatiky**

# Server pro podporu výuky ARUO

Supporting Server for Constraint Processing Teaching

2017/2018 Bc. Lukáš Nevřela

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

# Zadání diplomové práce

Student: **Bc. Lukáš Nevřela**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Server pro podporu výuky ARUO
Supporting Server for Constraint Processing Teaching

Jazyk vypracování: čeština

Zásady pro vypracování:

Cílem diplomové práce je vytvořit server, na kterém si studenti budou moci spustit výpočet vybraných algoritmů z oblasti constraint processing na jimi zadaných i ukázkových vstupech a na zobrazeném průběhu výpočtu lépe pochopit princip fungování těchto algoritmů. Tato práce se konkrétně zaměří backtracking a jeho možná vylepšení.

1. Nastudujte si algoritmus backtracking a možné algoritmy pro jeho urychlení (pro dopřednou fázi i zpětnou fázi).
2. Cílem není nejefektivnější implementace algoritmů, ale taková, která umožní zobrazovat postupně jednotlivé kroky tak, aby uživatel mohl pochopit princip jejich fungování.
3. Vytvořte výukový server, kde bude možné zadat vstupní data pro naprogramované algoritmy a pro tato data si nechat zobrazit postup výpočtu.
4. Vytvořte i ukázkové vstupy pro jednotlivé algoritmy, aby postup výpočtu bylo možné zobrazit i bez zadávání vstupu uživatelem.

Seznam doporučené odborné literatury:

[1] Rina Dechter: Constraint Processing, Morgan Kaufmann Publishers 2003, ISBN: 978-1-55860-890-0

Další literatura dle pokynů vedoucího diplomové práce.

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **Ing. Martin Kot, Ph.D.**

Datum zadání: 01.09.2015
Datum odevzdání: 30.04.2018

doc. Ing. Jan Platoš, Ph.D.
*vedoucí katedry*

prof. RNDr. Václav Snášel, CSc.
*děkan fakulty*

**Prohlášení**

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě dne 30. 4. 2018

.....
Podpis

## Poděkování

Na tomto místě bych rád poděkoval Ing. Martinu Kotovi, Ph.D. za jeho cenné rady a připomínky a vedení při vypracování této práce.

## Abstrakt

Cílem práce je nastudovat algoritmy řešící problémy s omezujícími podmínkami a vytvořit aplikaci, která bude demonstrovat jejich průběh a postup výpočtu. Výsledná aplikace umožňuje definovat vlastní zadání problému, který bude vyřešen vybraným algoritmem a uživateli bude zobrazen průběh výpočtu formou animace procházení stromové struktury prostoru potencionálních řešení. Práce je zaměřena na backtracking a jeho modifikace. Do aplikace byl implementován základní backtracking, backjumping, dopředná kontrola a hranová závislost. Je založena na webových technologiích HTML 5, CSS 3 a JavaScript. Aplikace bude nasazena na univerzitní server. Je určena studentům předmětu ARUO pro studium a lepší pochopení vybraných algoritmů a znázornění rozdílu mezi nimi.

## Klíčová slova

Backtracking, backmarking, dopředná kontrola, hranová závislost, backjumping, učící se algoritmy, grafově založené učení, dopředné algoritmy, zpětné algoritmy, proměnná, doména, omezení, síť omezení, minimální síť, zastavení, dopředná fáze, zpětná fáze, konfliktní množina.

## Abstract

The goal of the thesis is to study algorithms solving problems in a field of contrained conditions and to develop an application to demonstrate their processing and progression. The application will allow user to enter his own specification of the problem, choose an algorith to solve it and to display the process of the algorithm. Result is displayed as an animation of browsing of a tree structure representing the space of potential solutions. Thesis is dedicated to the backtracking and its modifications. Application implements basic backtracking, backjumping, forward-chacking and arc-consistency algorithms.It is ased on web technologies as HTML 5, CSS 3 and JavaScript. Application will be deployed on university server. It is accessible to the students of ARUO for studying and better understanding of the selected algorithms and to demonstrate differences among them.

## Key word

Backtracking, backmarking, forward-checking, Arc-consistency, backjumping, learning algorithm, graph based learning, look-ahead algorithm, look-back algorithm, variable, domain, constraint, constraint network, minimal network, dead-end, forward phase, backward phase, conflict set.

# Obsah

# Seznam použitých zkratek

AC - Arc Consistency

ARUO - Automatizované řešení úloh s omezením

CSS - Cascading Style Sheet

GUI - Graphic User Interface

HTML - Hyper Text Makeup Language

JSON - JavaScript Object Notation

PHP - Personal Home Page, Hypertext Preprocessor

# Seznam ilustrací

# Seznam tabulek

# 1 Úvod

Tato diplomová má za cíl vytvoření výukového serveru pro předmět Automatizované řešení úloh s omezením (ARUO). V rámci tohoto předmětu se řeší problémy, které spadají do oblasti constraint processing (úlohy s omezením). Vytvořený server se bude specializovat na pochopení backtrackingu a algoritmů, které zrychlují hledání řešení. Tato práce se dělí na dvě části: teoretickou, kde jsou popsány jednotlivé algoritmy, a praktickou část, ve které jsou popsány technologie, samotná implementace a testování aplikace.

Teoretická část práce je zpracována ve čtyřech kapitolách. Nejdříve je představen základní backtracking a jeho vylepšení backmarking. Následně jsou rozebrány algoritmy řešící dopřednou fázi backtrackingu, jako jsou dopředná kontrola a hranová závislost. V další kapitole jsou popsány algoritmy optimalizující zpětnou fází backtrackingu. Jedná se o backjumping a učící se algoritmy. V závěru této teoretické části jsou jednotlivé postupy porovnány.

Praktická část se zabývá popisem vytvořené aplikace. Nejprve je rozebrána architektura systému a technologie použité pro jeho implementaci. Další část je zaměřena na implementaci jednotlivých částí architektury. Poslední část je věnována testování fungování aplikace.

Na závěr je provedeno stručné shrnutí.

# 2 Backtracking

Constraint processing neboli řešení s omezením je problematika, která se zabývá nalezením řešení pro případy, kdy nalezení řešení závisí na omezení mezi jednotlivými prvky. Pracuje se v ní s trojicí množin X, D, C. Mezi způsoby nalezení řešení existuje algoritmus backtrackingu. Přičemž:

- X je množina jednotlivých proměnných. Těm se přiřazují hodnoty z jejich domén s cílem splnit všechna omezení.

- D je množina domén pro každou proměnnou. Každá proměnná *x* má přiřazenu jednu doménu *d*, v níž se nacházejí hodnoty, kterých může daná proměnná nabývat.

- C je množina všech omezení proměnných a platí mezi dvěma proměnnými. Jedná se o podmínky, které musí splňovat přiřazené hodnoty. Pokud budou některé hodnoty v rozporu s omezením, pak toto přiřazení hodnot není součástí řešení.

Backtracking je metoda založená na procházení stromové struktury do hloubky. Algoritmus vylepšuje hledání řešení hrubou silou (testování všech možných řešení) zavedením struktury do prostoru potencionálních řešení. Při jejím procházení je schopen vyloučit velké množiny řešení, aniž by je přímo zkoušel. Dělí se na dvě fáze: fázi dopřednou a fázi zpětnou. Nejprve se postupně přiřazuje jednotlivým proměnným hodnota z jejich domén. Po každém doplnění hodnoty zkontroluje, zda splňuje všechna omezení. Pokud přiřazení některé omezení nesplňuje, pak se této proměnné přiřadí další hodnota z její domény. Pokud byly vyzkoušeny všechny hodnoty z domény, je třeba se vrátit na předchozí proměnnou a této proměnné zkusit přiřadit novou hodnotu z její domény. Tento proces se nazývá zpětná fáze. Pokud jsou po dosazení hodnoty z domény splněna všechna omezení pro danou proměnnou, přejde se na další proměnnou a celý proces dosazování a kontroly probíhá znovu. Tento proces se nazývá dopředná fáze.

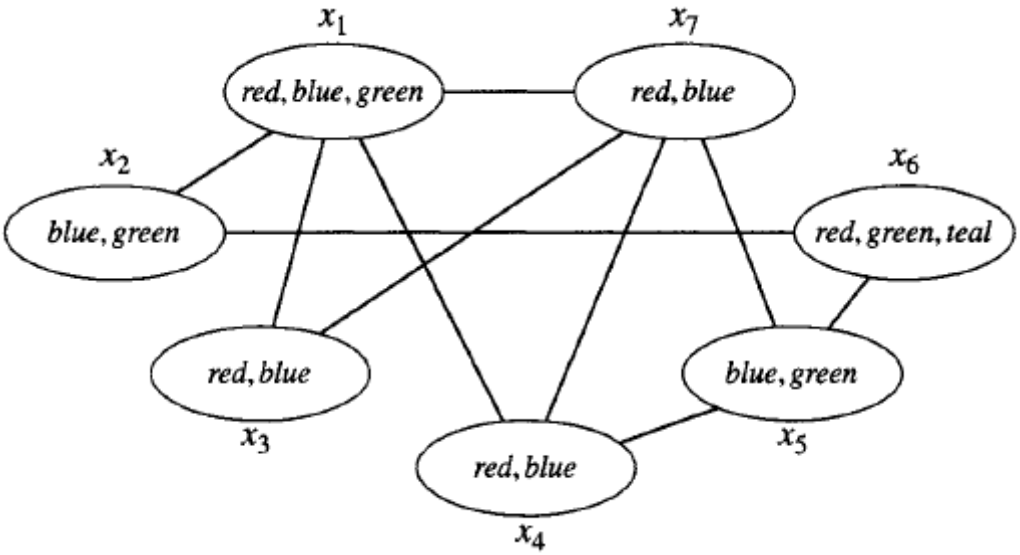
V případě, že dojde k nesplnění omezení a již nejsou žádné hodnoty k dosazení, jedná se o tzv. uváznutí (dead-end). Pokud se v doméně proměnné nacházejí další prvky, které by se mohly do proměnné dosadit, tak se přiřadí následující prvek a znovu se zkontrolují omezení. Jestliže dojde k uváznutí, přičemž se nenašla žádná proměnná, která by splňovala omezení, a zároveň byly všechny prvky proměnné přiřazeny, jedná se o listové uváznutí.

Řešení je nalezeno tehdy, kdy všechny proměnné mají přiřazenou nějakou hodnotu a jsou splněna všechna omezení. Ve většině případů po nalezení řešení se backtracking ukončí. Někdy je však požadováno, aby algoritmus nalezl několik nebo všechna řešení. V tom případě se uloží nalezené řešení – přiřazení hodnot pro každou proměnnou – a pokračuje se ve výpočtu. Algoritmus se chová jako při uváznutí a buď se proměnné přiřadí další hodnota z její domény, nebo pokud se již v doméně další hodnota nenachází, se vrací na předchozí proměnnou.

Jednou z možností, jak lze zobrazit průběh provedeného algoritmu je pomocí grafu, který bude obsahovat stromovou strukturu. Průchod algoritmu a jeho prohledávaný prostor se dá ovlivnit několika způsoby. Jeden z nich je změna pořadí proměnných, kterým se bude přiřazovat hodnota (viz příklad 1). Dále existují algoritmy, které pročešou stromovou strukturu a urychlí algoritmus backtrackingu pro některé případy při hledání řešení. Tyto algoritmy se dělí na dopředné (look ahead) a zpětné (look back).

**Příklad 1 [1]**

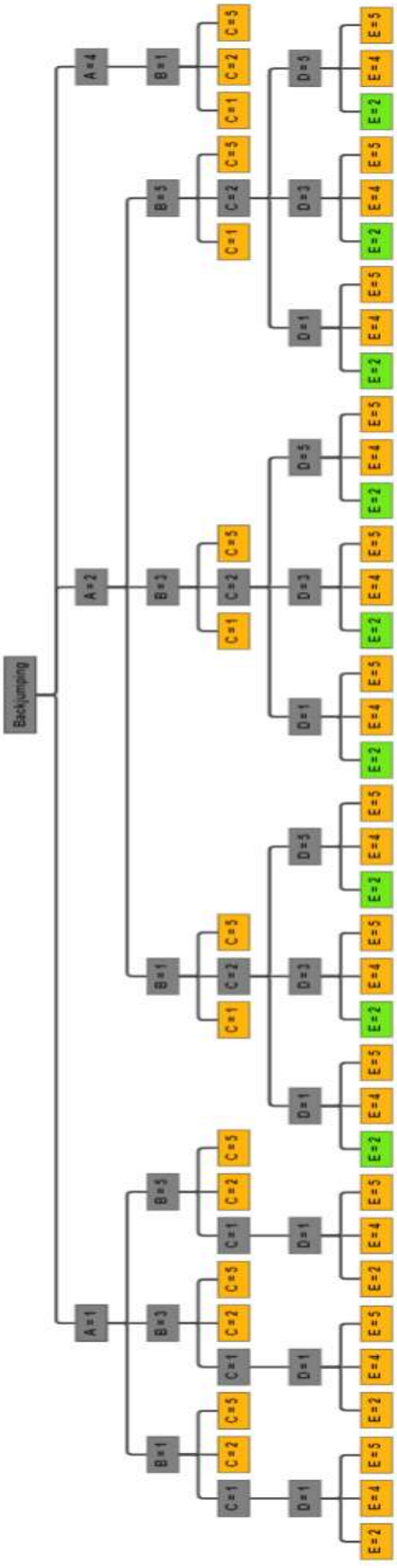
Jednoduchý příklad barvení grafu o sedmi vrcholech viditelný na obrázku 1




*Obr. 1 Graf s vypsanými barvami, kterých mohou vrcholy nabývat [1]*

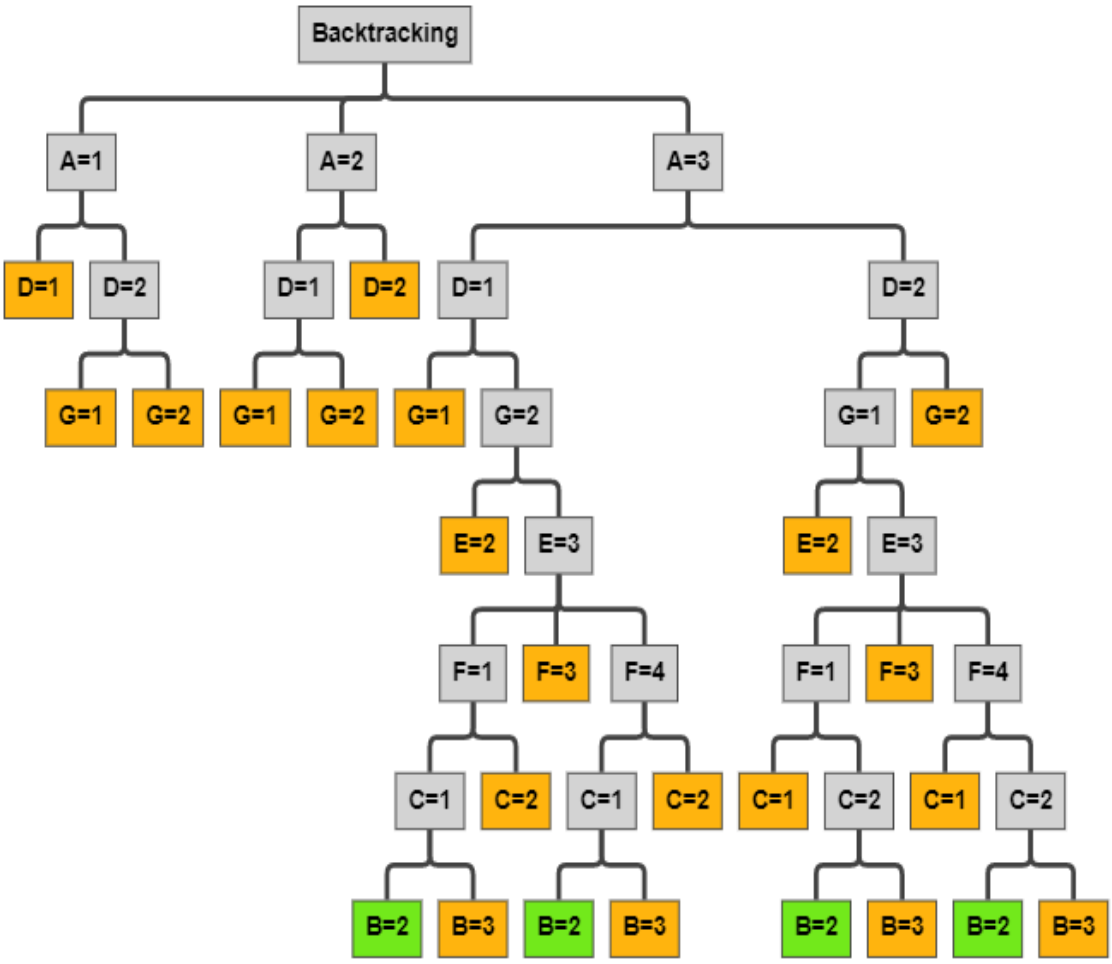
Jako proměnné jsou zvoleny vrcholy $x_1$-$x_7$, domény pro jednotlivé proměnné jsou jednotlivé barvy, kterými mohou být vrcholy zabarveny – červená (red), modrá (blue), zelená (green) a tyrkysová (teal). Omezení budou představovat hrany mezi jednotlivými vrcholy (proměnnými). Omezením je, že barva jednoho vrcholu se nesmí shodovat s barvou druhého vrcholu. Z tohoto zadání vznikne následující stromová struktura zobrazena na obrázku č. 2. Šedé uzly jsou rozhodující body, zelené uzly jsou nalezená řešení, oranžové uzly jsou uváznutí algoritmu.

Na obrázku č. 2 je možné sledovat průběh backtrackingu, kde čísla představují jednotlivé barvy (1 – red, 2 – blue, 3 – green, 4 - teal). Nejprve se provede dopředná fáze, kdy se přiřadí hodnota pro proměnnou $x_1$ a zkontrolují se omezení. Pokud se žádná omezení neporušila, hodnota proměnné $x_1$ splňuje omezení a začne se přiřazovat hodnota pro následující proměnnou $x_2$. Takto bude proces probíhat až po proměnnou $x_6$, kdy dojde k nesplnění omezení. Začne zpětná fáze, průběh výpočtu se vrátí zpět na předchozí proměnnou a poté bude pokračovat opět dopřednou fází. Proměnné $x_5$ se přiřadí následující hodnota z domény, která ještě nebyla přiřazena. Při přiřazení (1-3-2-2-3-4-2) si lze povšimnout, že při zpětné fázi se proces vrací zpět až na proměnnou $x_1$, a to proto, že předchozí proměnné již neměly žádnou další hodnotu v doméně nebo žádná z těchto hodnot nesplňovala omezení.




*Obr. 2 Stromová struktura backtrackingu pro příklad 1*

Na obrázku č. 3 je znázorněno, jak se změní tato struktura, pokud se změní pořadí proměnných, kterým se budou přiřazovat hodnoty.




*Obr. 3 Stromová struktura backtrackingu pro příklad 1 se změněným pořadím přiřazovaných proměnných*

## 2.1 Vylepšení backtrackingu

Základní backtracking má několik nedostatků, přičemž nejzásadnějším je dlouhá doba hledání. Všechny nedostatky se nedají úplně odstranit, ale pomocí různých heuristik je možné jejich vliv minimalizovat. Podle oblasti optimalizace backtrackingu se heuristiky dělí do dvou kategorií: na dopředné algoritmy (look-ahead) a zpětné algoritmy (look-back). Look-ahead algoritmy se zabývají, jak je z názvu patrno, dopřednou fázi backtrackingu, zatímco look-back algoritmy se zabývají zpětnou fází. Obě kategorie vylepšení jsou podrobně rozebrány v následujících kapitolách této práce.

Existují i další vylepšení backtrackingu. Příkladem může být backmarking. Jedná se o jednu z prvních optimalizací vylepšujících průběh hledání řešení. Backmarking však nezapadá zcela do look-ahead algoritmů ani do kategorie look-back algoritmů.

## 2.2 Backmarking

Tento algoritmus funguje tak, že si zaznamenává prošlé kontroly omezení, a díky tomu je backmarking dokáže odstranit, aniž by se musely opakovat znovu stále stejné kontroly.

Předpokládá se, že pro současnou proměnnou $x_i$ a její předešlou proměnnou $x_j$ platí: pokud tyto dvě proměnné budou mít přiřazeny svoje hodnoty, které již měly aspoň jednou přiřazené, a proběhla na nich kontrola, omezení vrátí vždy stejný výsledek - omezení je splněno anebo naopak splněno není. Díky zaznamenávání této informace se může okamžitě zjistit, jestli jsou hodnoty z domény $x_i$ konzistentní nebo nekonzistentní. Přiřazení, která se budou muset kontrolovat, jsou tedy jenom ta, která ještě nebyla kontrolována.

Backmarking upravuje backtracking tím, že si dodatečné informace uchovává ve dvou tabulkách. V první tabulce jsou uloženy všechny hodnoty pro proměnnou $x_i$ a její hodnoty $a_v$. Backmarking si pamatuje nejzazší proměnnou $x_j$ takovou, že současné přiřazení je v konfliktu se současnou proměnnou $x_i$. Do ní je zaznamenána hodnota $M_{i,v}$ – dvojice, kde $i$ je číslo proměnné a $v$ je její hodnota. Do druhé tabulky zaznamenává $low_i$. Hodnota $low_i$ označuje nejnovější proměnnou, která změnila svou hodnotu od okamžiku, kdy byla naposledy přiřazena hodnota proměnné $x_i$. Pokud je záznam z první tabulky $M_{i,v}$ menší než $low_i$, algoritmus pozná, že se hodnota nezměnila, a tudíž znovu nesplní omezení. Pokud bude hodnota větší nebo rovna $low_i$, pak omezení splní. V obou případech se nemusí provádět kontrola omezujících podmínek.[1]

Na příkladu při první situaci, kdy dojde k uváznutí, jsou proměnným přiřazeny následující hodnoty ($x_1$, blue), ($x_2$, green), ($x_3$, red), ($x_4$, red), ($x_5$, green), ($x_6$, red). Do první tabulky se zapíšou pomocné hodnoty: $M_{(1, blue)} = 1$, $M_{(2, green)} = 2$, $M_{(3, blue)} = 1$, $M_{(3, red)} = 3$, $M_{(4, blue)} = 1$, $M_{(4, red)} = 4$, $M_{(5, blue)} = 5$, $M_{(6, green)} = 2$, $M_{(6, red)} = 6$, $M_{(7, red)} = 3$, $M_{(7, blue)} = 1$. Při backtrackingu z $x_7$ na $x_6$, $low(7) = 6$ and $x_6$ je přiřazena hodnota teal. Při pokusu o přiřazení nové hodnoty pro $x_7$ algoritmus zkontroluje pomocné údaje. Zjistí, že hodnota M je menší než $low_7$ pro obě hodnoty, a proto je schopen rozhodnout, že přiřazení nesplňuje omezení, i bez použití kontroly omezení.

Backmarking stromovou strukturu tedy nijak nezredukuje, ale odstraní množství kontrol omezení ukládáním pomocných informací o předchozích vyhodnocováních potencionálních řešení problému.

# 3 Dopředné algoritmy (Look-ahead)

Dopředné algoritmy se zabývají optimalizací dopředné fáze backtrackingu. Algoritmy se snaží zjednodušit prohledávaný prostor tím, že sníží počet hodnot v doménách proměnných před samotným backtrackingem. Nebo se také provádějí v momentu před přiřazením hodnoty proměnné. Rozhodují o tom, jakou hodnotu přiřadit na základě toho, jak přiřazení ovlivní složitost struktury. Všechny tyto optimalizace mohou, pokud není pevně stanoveno, také měnit pořadí, ve kterém se budou přiřazovat proměnné.

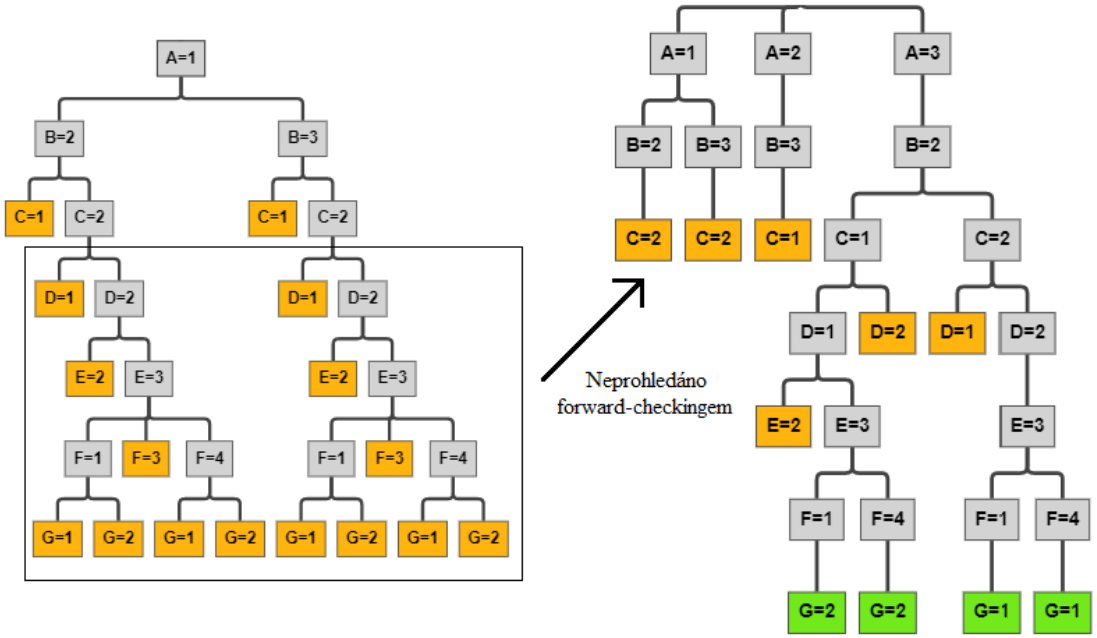
Stěžejní úlohu hraje schopnost posoudit, do jaké hloubky se bude provádět analýza stromové struktury potenciálních řešení. To se provádí ještě před přiřazováním hodnot proměnným, které bude trvat určitý čas. Je proto nutné nalézt rovnováhu mezi dobou hledání možností, jak zjednodušit strukturu možných řešení, proti době samotného procházení struktury a vyhodnocování jednotlivých uzlů. Zjednodušení se může týkat toho, že se pro právě přiřazovanou proměnnou pomocí dopředných algoritmů ukáže, jestli je některá doména dalších proměnných prázdná. Pak stávající dosazení proměnných se nenachází v řešení a může se rovnou provést zpětná fáze. Díky tomu dochází k uváznutí algoritmu dříve a zjednoduší se struktura, kterou je třeba prohledat. V případě, že odstraníme z domén hodnoty, které nemohou nastat kvůli nesplnění omezení, ještě před samotným backtrackingem, zjednoduší se samotné výpočty algoritmů pro dopřednou fázi.

Tyto algoritmy jen výjimečně zlepší výkonnost backtrackingu v nejhorším případě, proto je tedy správné zajistit správné vyvážení.

## 3.1 Dopředná kontrola (forward-checking)

Tento algoritmus provádí méně rozsáhlé šíření omezení. Zkoumá, zda vybraná hodnota je nezávadná pro každou budoucí proměnnou zvlášť[1]. Pokud se při tomto prohledávání zjistí, že prohledávaná doména je prázdná, pak hodnotu, pro kterou se toto prohledávání provádělo, nevybere a zvolí další, pro kterou tento postup opakuje.

Na příkladu barvení grafu (obrázek 4) lze vidět, že po dosazení hodnoty red pro proměnnou $x_1$ se tato hodnota odstranila z domén proměnných $x_3$, $x_4$ a $x_7$. Při dosazení hodnoty proměnné $x_2$ se nic nezmění. Pro proměnnou $x_3$ je možné dosadit pouze hodnoty red a blue, ovšem hodnota red se z domény odstraní při dosazení proměnné $x_1$ a tudíž zbývá hodnota blue, ovšem při tomto dosazení se odstraní hodnota blue z domény proměnné $x_7$ a tím doména této proměnné bude prázdná. Toto dosazení hodnot proto nevede k řešení. Z toho důvodu provede algoritmus simulaci uváznutí a provede zpětnou fázi.




*Obr. 4 Výsek stromové struktury ukazující princip dopředné kontroly*

## 3.2 Hranová závislost (Arc consistency)

Algoritmus arc consistency se liší od algoritmu forward-checking tím, že se provádí výpočty na všech ještě nepřiřazených proměnných. Opět platí, že pokud se při výpočtech objeví prázdná doména, pak se pro toto přiřazení nenachází řešení a současná hodnota je zamítnuta.

Hranová závislost funguje na principu vyřazení hodnot z domén, pro které nelze přiřadit další hodnotu. Pokud proměnná $x_1$ bude mít doménu hodnot $d_{x1} = [1, 2, 3, 5, 8]$, pak se vybere hodnota 1. Pro tuto hodnotu se nyní hledá, jestli existuje nějaká hodnota v doméně jiné proměnné, se kterou má stávající proměnná nějaké omezení. Jestli proměnná $x_2$ bude mít doménu hodnot $dx_2 = [1, 2, 4, 8]$ a jestli omezení mezi proměnnými $x_1$ a $x_2$ bude definováno jako jejich rovnost ($x_1=x_2$), pak pro prvek 1, který se nachází v obou proměnných a splňuje tedy omezení, se ponechá v doméně $x_1$. Totéž pro prvky 2 a 8. Avšak pro prvek 3, který se nenachází v doméně $dx_2$ a tudíž nesplňuje omezení, se musí prvek odstranit. To stejné platí pro prvek 5. Po provedení hranové závislosti pro proměnnou $x_1$ bude její doména vypadat takto: $dx_1 = [1, 2, 8]$. Existuje několik algoritmů arc consistency, které se liší asymptotickou složitostí a složitostí na naprogramování:

Arc consistency 1 (AC-1). Tento algoritmus funguje na principu brute-force, kdy se vyzkouší všechny kombinace hodnot všech proměnných. Pokud některá dvojice nesplňuje omezení, odstraní hodnotu právě prohledávané proměnné. Pokud se odstraní některá hodnota, projdou se znovu všechny proměnné. Algoritmus končí v případě, že se během iterace prohledávání neodstranila žádná hodnota a může se pokračovat v backtrackingu, nebo v případě, že některé doméně se odeberou všechny hodnoty a stane se prázdnou. To znamená, že neexistuje žádné řešení pro tento problém.

Arc consistency 3 (AC-3). Tento algoritmus funguje na stejném principu jako AC-1, přidává však další optimalizaci. Udržuje si seznam proměnných, které je potřeba projít. Pokud se změnila některá proměnná, pak se projdou jenom ty proměnné, které mají s procházenou proměnnou společné některé omezení. Algoritmus AC-2 je mezikrok mezi AC-1 a AC-3.

Arc consistency 4 (AC-4). Algoritmus AC-3 se dá upravit pomoci AC-4 z hlediska časové náročnosti. Arc consistency 4 přiřadí každé hodnotě $a_i$ z proměnné $x_i$ všechny konzistentní hodnoty z proměnné $x_j$. Hodnota $a_i$ je odebrána pokud nemá žádnou konzistentní hodnotu s některou z proměnných. Tento algoritmus se také nazývá opravdový úplný dopředný pohled (real full look-ahead). To proto, aby se odlišil od plného a částečného dopředného pohledu (full/partial look-ahead).

Velmi často je využívána metoda Údržba hranové závislosti (Maintaining-Arc-Consistency – MAC). Tato metoda funguje na principu, že při vyřazení hodnoty se provede znovu úplná hranová závislost, tedy na principu AC-1.

Arc consistency 6 (AC-6). Algoritmus AC-6 je spojením algoritmů AC-3 a AC-4. Algoritmus se liší od AC-4 tím, že taky má přiřazeno ke každé hodnotě $a_i$ z proměnné $x_i$ její konzistentní hodnoty, avšak v tomto případě se nestará o celkový počet, ale zajišťuje, že má aspoň jednu konzistentní hodnotu. Tudíž AC-6 potřebuje pouze seznam $S[x_i, a_i]$, který obsahuje všechny hodnoty, pro které $(x_i, a_i)$ je současná konzistentní[3].

Arc consistency 2001 (AC-2001). Tento algoritmus je podobná AC-3, ale dosahuje optimalizace ukládáním nejmenší konzistence pro každé omezení jako AC-6. Rozdíl je v tom jak je tato informace uložena na rozdíl od AC-6. AC-2001 nepoužívá seznam obsahující konzistentní hodnoty, ale používá ukazatele $Last[x_i, a_i, x_j]$ pro $a_j$.[3]

## 3.3 Závislost cesty (Path consistency)

Algoritmus path consistency je velmi podobný algoritmu arc consistency. U arc consistency, aby hodnota z domény nebyla vyřazena tak musí existovat přiřazení nějaké předchozí proměnné, které bude splňovat podmínky. Algoritmus path consistency funguje na stejném principu, avšak místo přiřazení jedné proměnné je potřeba přiřazení dvou proměnných

.

## 3.4 i-závislost (i-consistency)

Tento algoritmus funguje na stejném principu jako předchozí algoritmy (arc a path consistency). Avšak počet přechozích proměnných, které musí být přiřazeny, jsou rovny $i$. Z toho vyplývá, že algoritmus pro i=2 je path consistency, pro i=1 je to arc consistency a pro i=0 se jedná o obyčejný backtracking.

## 3.5 Backtracking s náhodným výběrem

Tento algoritmus funguje na stejném principu jako obyčejný backtracking avšak při výběru prvku z domény se prvky nepřiřazují postupně, ale vybere se náhodný prvek, který se proměnné přiřadí.

## 3.6 Dynamické řazení proměnných (Dynamic value ordering)

Tento algoritmus funguje tak, že se proměnné seřadí podle velikosti svých domén od nejmenší po největší. Posléze se provádí hledání řešení pomocí algoritmu backtrackingu. A to z toho důvodu, že takto vznikne nejmenší stromová struktura. U obyčejného backtrackingu se proměnné seřadí před algoritmem a dále se již pořadí nemusí měnit, jelikož se nemění jejich domény. Avšak u algoritmu, který za běhu upravuje velikost domén jako je např. forward-checking, tak se po každé úpravě domén kontroluje, jestli není potřeba změnit pořadí zbylých proměnných.

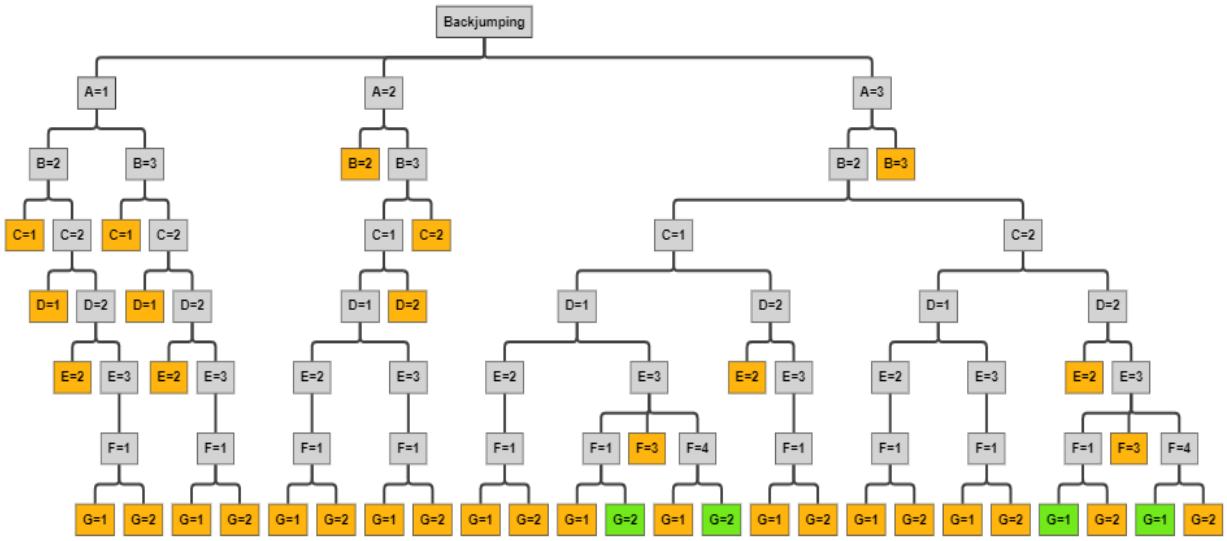
# 4 Zpětné algoritmy (Look-back)

Tyto algoritmy se specializují na zrychlení samotného backtrackingu řešením tzv. trashingu. Trashing znamená, že se testují kombinace, které nesplňují omezení a již byly dříve testovány. Look-back algoritmy se využívají při zpětné fázi. Dělí se na dvě skupiny: zabývající se zrychlením zpětné fáze a zabraňující procházení stejných konfliktních situací.

## 4.1 Backjumping

Backjumping algoritmy se specializují na zrychlení zpětné fáze backtrackingu. Při zpětné fázi se nejde zpět pouze o jeden krok, ale analyzuje se uváznutí a podle situace se vrátí zpět až na konfliktní proměnnou. Příkladem backjumping algoritmů jsou algoritmy zmíněny níže. Nejedná se o kompletní výpis, existují i další. Na obrázku č. 5 lze vidět vykreslený graf pro backjumping na uvedeném příkladu.

- Gashnigův backjumping (Gashnig's backjumping)
- Grafově založený backjumping (Graph based backjumping)
- Konfliktně řízený backjumping (Conflict-directed backjumping)

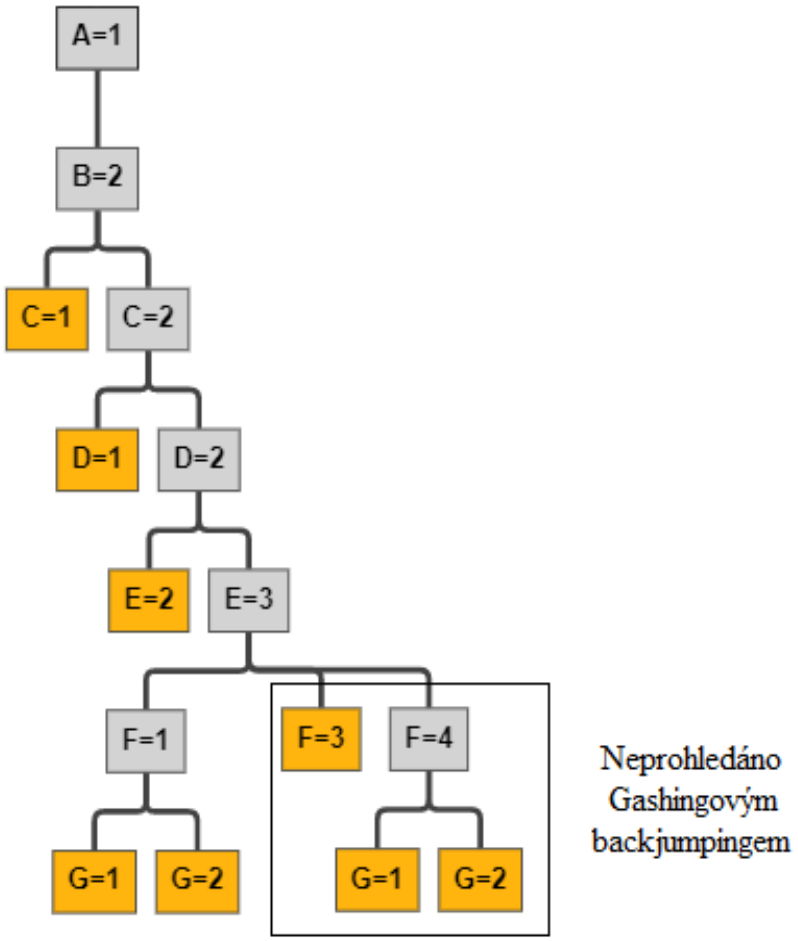



*Obr. 5 Stromová struktura příkladu 1 ukazující možné zpětné skoky*

### 4.1.1 Gashnigův backjumping (Gashnig's backjumping)

Gashnigův backjumping se zabývá skoky zpět založenými na závadné proměnné. Závadná proměnná je taková proměnná, jejíž hodnota zapříčiňuje uváznutí. Zpětné skoky o více proměnných u tohoto algoritmu se provádějí pouze z listového typu uváznutí. Například větev nejvíce nalevo v obrázku č. 5, která je zobrazena na obrázku č. 6, kde je možné vidět, že po dosazení proměnné $x_6$ nastane listové uváznutí. Podle omezení lze zjistit, že závadná proměnná je $x_3$.

Na rozdíl od klasického backtrackingu se proto nevrací pouze o jeden krok zpět k proměnné $x_5$, ale skočí až na závadnou proměnnou $x_3$. Jelikož hodnota proměnné $x_6$ bude vždy v konfliktu s hodnotou proměnné $x_3$, tak nezáleží na tom, jaké hodnoty se doplní pro proměnné $x_4$ a $x_5$, jelikož to vždy skončí na konfliktu $x_3$ a $x_6$. Tudíž lze bezpečně skočit zpět až na tuto proměnnou. Nesmí se ovšem skočit na předchozí proměnnou $x_2$, jelikož následující hodnota proměnné $x_3$ by se nemusela nacházet v konfliktu s hodnotou proměnné $x_6$ a mohlo by se přeskočit hledané řešení problému.




*Obr. 6 Ukázka Gaschnigova backjumpingu*

### 4.1.2 Grafově založený backjumping (Graph based backjumping)

V Gashnigově backjumpingu se prováděl backjumping pouze v případě, kdy se dospělo k listovému typu uváznutí. Při vnitřním uváznutí by se tento backjumping choval jako klasický backtracking a došlo by k navrácení zpět pouze na předchozí proměnnou. Grafově založený backjumping provádí backjumping u listových uváznutí, ale také u těch vnitřních. Tento algoritmus získává informace o možných konfliktních množinách ze struktury grafu[1].

Konfliktní množina je definována následujícím způsobem. Nechť $\bar{a} = (a_{i1},\ldots,a_{ik})$ je konzistentní dosazení hodnot libovolné podmnožiny proměnných a $x$ je dosud nedosazená proměnná. Pokud v doméně $x$ neexistuje hodnota $b$ taková, aby $(\bar{a},x=b)$ bylo konzistentní, je možné nazvat $\bar{a}$ konfliktní množinou (conflict set) proměnné $x$. Pokud navíc $\bar{a}$ neobsahuje žádnou podmnožinu konfliktní s $x$, je

možné nazvat $\bar{a}$ minimální konfliktní množinou proměnné $x$[2]. Jinými slovy, pokud je již několik proměnných přiřazených a potřebuje se přiřadit hodnota k proměnné $x$ a neexistuje žádná hodnota, která by splňovala všechna omezení, jsou tyto hodnoty proměnné konfliktní množina.

Pokud při průchodu stromovou strukturou dojde k jakémukoli typu uváznutí, pak se při grafově založeném backjumpingu zjistí minimální konfliktní množina a vrátí se zpět na její poslední prvek. Pokud u této proměnné již nebude možnost přiřadit další hodnotu, provede se backjumping a opět se skočí na poslední přiřazenou hodnotu této proměnné. Pokud se vyjde ze stejného příkladu a vezme se prostřední větev z obrázku č. 5 větev výpočtu, pak při přiřazení hodnot pro proměnnou $x_7$ dojde ke konfliktu a pomocí backjumpingu se skočí na proměnnou $x_5$. Této proměnné by se přiřadila nová hodnota a pokračovalo by se dopřednou fází backtrackingu. Avšak v případě, kdy by již nebylo možné přiřadit novou hodnotu proměnné $x_5$, by se opět provedl backjumping s předpokladem, že by mezi proměnnými $x_3$ a $x_5$ nebylo žádné omezení, ale bylo by mezi proměnnými $x_1$ a $x_3$. V tomto případě by se provedlo vrácení z proměnné $x_5$ až na proměnnou $x_1$.

Grafově založený backjumping používá podmnožinu předchozích přiřazených proměnných k proměnné $x$ jako aproximaci minimální konfliktní množiny $x$. Je aproximační, protože i když existuje omezení mezi proměnnými $x$ a $y$, konkrétní hodnota přiřazená proměnné $y$ nemusí být v konfliktu s hodnotou proměnné $x$. Například v uvedeném příkladu pro proměnnou $x_2$ hodnota blue nemá vliv na proměnnou $x_6$, jelikož se hodnota blue v doméně proměnné $x_6$ nevyskytuje. Protože grafově založený backjumping neudržuje informace o hodnotách domén, předpokládá nejhorší případ, pro který je podmnožina proměnných připojených k $x$ minimální konfliktní množina $x$. S předpokladem je poslední přiřazená proměnná, která předchází $x$, závadná proměnná[1].

### 4.1.3 Konfliktně řízený backjumping (Conflict-directed backjumping)

Předchozí verze algoritmu backjumping fungovaly na principu, že se provede návrat zpět na konfliktní proměnnou, nebo že se vrátí o malý skok na poslední přiřazenou proměnnou, se kterou je současná proměnná v konfliktu. Konfliktně řízený backjumping funguje na podobném principu jako grafově založený. Pomocné informace však nebere pouze z grafu, ale shromažďuje je během celého hledání[1]. Pro každou proměnnou udržuje tzv. skokovou množinu. Nejprve se definuje pořadí mezi omezeními.

Skoková množina je definována následovně:

- Skoková množina $J_{i+1}$ listového uváznutí $x_i$ je jeho *var-emc*($\bar{a}_i$).
- Skoková množina vnitřního stavu ā (nebo proměnné $x_{i+1}$) obsahuje všechny *var-emc*($\bar{a}_j$) všech relevantních uváznutí $\bar{a}_j$, $j \geq i$, které nastaly v současné řešení proměnné $x_i$.

Nechť pro problém R = (X, D, C) s uspořádáním proměnných $d$ je $\bar{a}_i$ n-tice, jejíž potenciální uváznutelná proměnná je $x_{i+1}$. Nejzazší minimální konfliktní množina pro $\bar{a}_i$ (nebo pro $x_{i+1}$), označovaná emc($\bar{a}_i$), může byt vytvořena takto: uvažujme omezení uspořádané podle relace dřívějšího, pro $j \in \{1,\ldots,$ c$\}$ (počet omezeni), pokud existuje $b \in D_{i+1}$ takové, že $R_j$ je porušeno přiřazením $\bar{a}_i$, $x_{i+1} = b$ a žádné dřívější omezeni tímto porušeno není, potom do *var-emc*($\bar{a}_i$) přidáme $S_j$ (rozsah $R_j$), *emc*($\bar{a}_i$) a získáme projekci $emc(\bar{a}_i) = a_i\,[var{-}emc(\bar{a}_i)]$[2].

Například proměnné pořadí ($x_1$,…,$x_9$), když omezení $R_1$ je ($x_3$, $x_5$, $x_8$, $x_9$) a omezení pro $R2$ je ($x_2$, $x_6$, $x_8$, $x_9$), pak $R_1$ je dříve než $R_2$, protože $x_5$ předchází $x_6$. Pro dané pořadí všech proměnných z množiny $X$ platí, že dřívější vztah mezi proměnnými definuje celkové pořadí pro omezení $C$.

Pro danou n-tici $\bar{a}$ je poslední proměnná ve skokové množině $J_i$ nejzazší proměnnou, kam je možné se bezpečně vrátit[2].

## 4.2 Učící se algoritmy (learning algorithm)

Předchozí algoritmy backjumpingu se pokoušely optimalizovat zpětnou fázi backtrackingu tím, že provedly několik skoků zpět. Učící algoritmy se snaží zjednodušit stromovou strukturu potenciálních řešení problému způsobem, že se budou stále učit (přidávat) nová omezení. Nejzazší minimální konfliktní množina se nazývá nedobrá a během prohledávání slouží jako zjištění jak se má provést backjumping. Taková nedobrá přiřazení se mohou zapsat jako nová omezení. Díky tomu se nebude možné dostat do stejného přiřazení proměnných, a proto se zjednoduší stromová struktura. Příležitost naučit se nové omezení nastane v situaci, kdy dojde k uváznutí. Pokud pro pouze současné přiřazení je $\bar{a}_i$ konfliktní množina pro $x_{i+1}$, je zbytečné přidávat tento konflikt mezi omezení. Pokud ovšem $\bar{a}_i$ obsahuje podmnožiny, které jsou v konfliktu s $x_{i+1}$, tak může být vhodné tento konflikt zařadit mezi omezení. Zařazením těchto malých omezení "prořeže" strom a urychlí tak backtracking tím, že se nebudou muset prohledávat stále stejná konfliktní přiřazení.

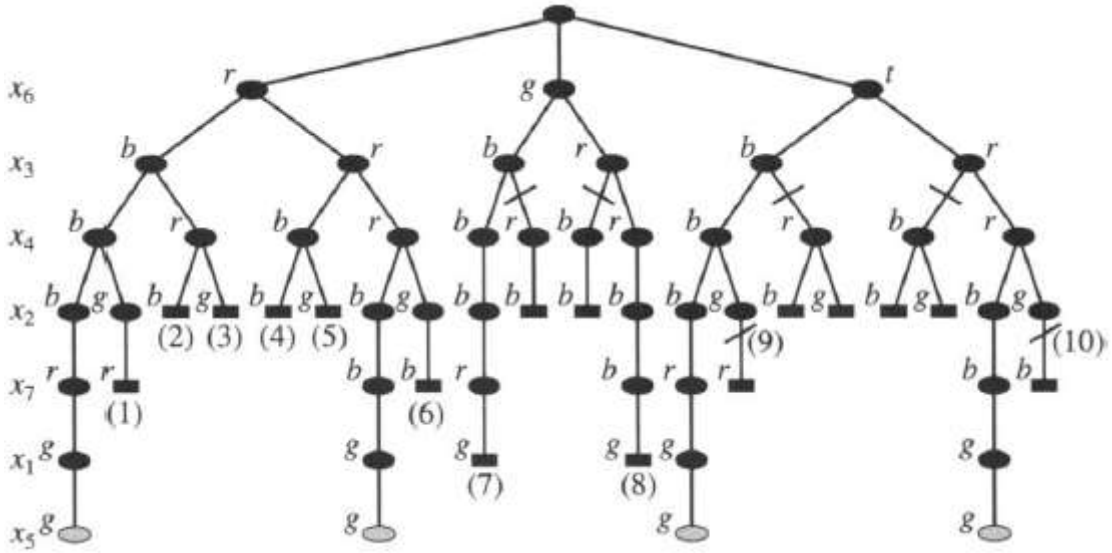
Pokud pro přiřazení proměnné $x_1 = 1$ a přiřazení proměnné $x_3 = 4$ nastane uváznutí, pak se tato situace nepřiřazuje mezi nová omezení. Pokud však pro proměnnou $x_1$ bude přiřazení proměnných $x_3 = 2$ a $x_4 = 8$ a pro tato přiřazení bude proměnná $x_6$ vždy v rozporu s proměnnými $x_3$ nebo $x_4$, pak se již vyplatí toto omezení přidat mezí nová omezení. Mezi nová omezení se přidají hodnoty zakázaných dvojic pro všechny $x_1$, že proměnná $x_3 \neq 2$ a zároveň $x_4 \neq 8$. Protože proměnná $x_1$ má podmnožiny proměnných v konfliktu, pak se toto omezení mohlo přidat.

Mezi často přidávaná omezení jsou prvky nejzazší minimální konfliktní množiny, která byla využívána při konfliktně založeném backjumpingu. V případě grafově založeného backjumpingu pak může být identifikována konfliktní množina a zařazena mezi omezení.

Učení nových omezení může být buď hluboké (deep) nebo povrchní (shallow). Deep-learning zařazuje mezi nová omezení pouze minimální konfliktní množiny. Shallow-learning naproti tomu umožňuje zařadit i neminimální konfliktní množiny. Deep-learning má tu nevýhodu, že jestli se jedná o minimální konfliktní množinu, je třeba použít náročnou analýzu.

### 4.2.1 Grafově založené učení (Graph based learning)

Grafově založené učení využívá stejných metod jako grafově založený backjumping.

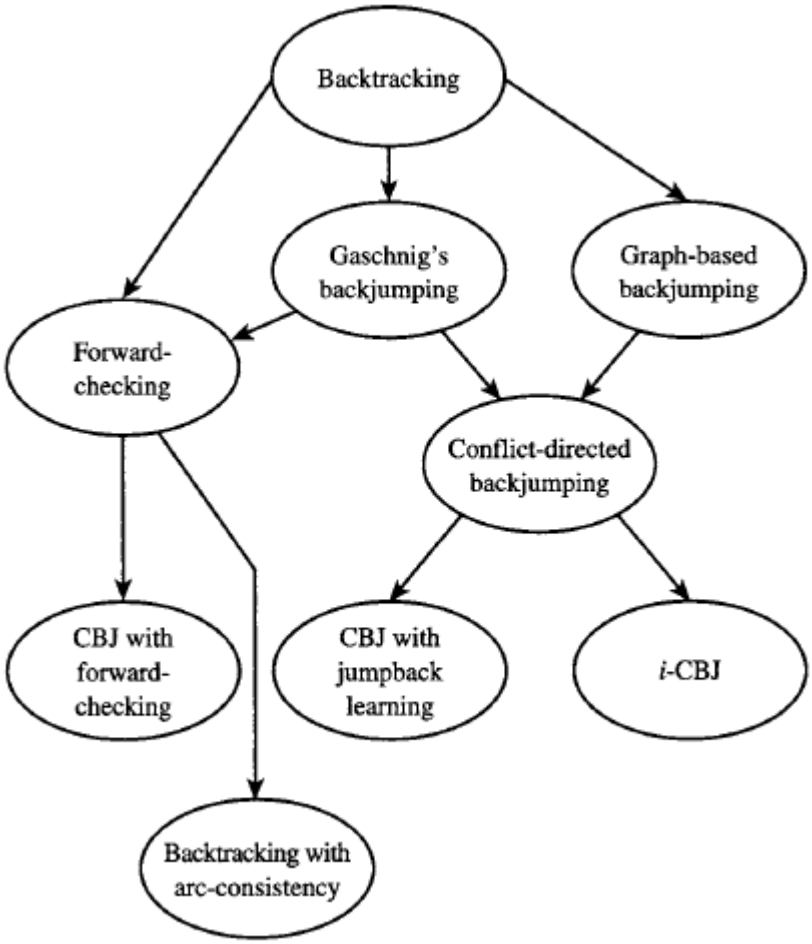



*Obr. 7 Stromová struktura příkladu 1 s ukázky možných škrtů větví, které se nebudou procházet [1]*

Z uvedeného příkladu při hledání všech řešení podle pořadí $x_6$, $x_3$, $x_4$, $x_2$, $x_7$, $x_1$ a $x_5$ je možné na obrázku 7 vidět hledání řešení pomocí klasického backtrackingu a backtrackingu využívající grafově založené učení. Přeškrtnuté větve jsou prohledávány pomocí klasického backtrackingu, ovšem nejsou prohledávány pomocí backtrackingu s učícím se algoritmem. Čísla 1 až 10 představují situace, kdy dojde k listovému uváznutí. Pro uváznutí 1 neexistuje žádná hodnota, která splňuje omezení. Předkové této proměnné jsou $x_2$, $x_4$, $x_3$ a $x_6$, proto grafově založené učení zaznamená nedobré hodnoty $x_2$-green, $x_4$-blue, $x_3$-blue *a* $x_6$-red. Tyto nedobré hodnoty se znovu opakují později při prohledávání a mohou být využity k "prořezání" podmnožiny vedoucí k uváznutí 9. uváznutí označeny jako 2 a 4 nastaly, jelikož se nepřiřadila žádná hodnota, která by splňovala omezení s proměnnou $x_7$, která má předky $x_3$ a $x_4$. Nedobré hodnoty $x_3$-blue, $x_4$-red a $x_3$-red, $x_4$-blue jsou zaznamenány grafově založeným učením pomocí vytvoření rovnosti mezi $x_3$ a $x_4$. Uváznutí označená jako 3 a 5 zahrnují stejné nedobré proměnné. Pokud by se využíval grafově založený backjumping, tyto uzly by se přeskočily. Tyto naučené omezení "prořežou" strom na šesti místech[1].

# 5 Porovnání algoritmů

V předchozích kapitolách bylo vysvětleno fungování backtrackingu a jeho vybrané optimalizace, která zlepšují jeho vyhledávání z několika hledisek. Ne všechny algoritmy zde byly probrány. Existuje jich více – jak pro dopřednou, tak i pro zpětnou fázi. Důležitá otázka zní, který nebo které z nich jsou nejlepší. To záleží na tom, podle čeho se budou algoritmy porovnávat. Srovnání mohou být pro nejhorší případ, průměrný případ atd. Na obrázku č. 8 jsou zobrazeny algoritmy, které jsou srovnány podle velikosti prohledaného prostoru [1]. Algoritmy jsou seřazeny tak, že algoritmus je efektivnější vůči svému předchůdci. Tedy grafově založený backjumping je efektivnější než klasický backtracking a je méně efektivní ve srovnání s konfliktně založeným backjumpingem.




*Obr. 8 Ukázka srovnání vyhledávacích algoritmů podle procházeného prostoru [1]*

Pro srovnání několika základních algoritmů proběhl experiment, při kterém se řešil jeden ZEBRA problém. Tento problém obsahoval dohromady 25 proměnných, mezi kterými bylo celkově 122 omezení. Toto umožnuje vytvoření několika milionů permutací problému. Bylo vybráno 450 problémů, kdy inicializace žádných dvou problémů se nerovnala, tudíž všech 450 problémů bylo unikátních.

Při zjišťování výsledku experimentu se nejprve zjišťoval počet provedených kontrol omezení. V následujícím kroku se zjišťovalo, kolik každý algoritmus provedl přiřazení jednotlivé proměnné, tedy kolik algoritmus provedl kroků. A nakonec se zohlednil čas pro vykonání problémů.

| Algo-ritmus | Kontroly omezení | | | Přiřazení hodnot | | | Doba výpočtu | | |
|---|---|---|---|---|---|---|---|---|---|
| | μ | Min | Max | μ | Min | Max | kontrol/sekunda | μ | Max |
| BT | 3 858 989 | 1773 | 102 267 383 | 746 728 | 321 | 15 862 302 | 11 973 | 322 | 8 541 |
| BJ | 503 324 | 358 | 19 324 081 | 92 842 | 68 | 3 200 564 | 8 418 | 59,8 | 2 295 |
| CBJ | 63 212 | 339 | 3 297 304 | 11 106 | 64 | 521 643 | 7 346 | 8,6 | 488 |
| BM | 396 945 | 401 | 18 405 514 | 746 728 | 321 | 15 862 302 | 1 151 | 344 | 15 990 |
| BMJ | 124 474 | 300 | 5 214 608 | 92 842 | 68 | 3 200 564 | 3 806 | 32,9 | 1 370 |
| BM-CBJ | 25 470 | 297 | 1 237 283 | 11 106 | 64 | 521 643 | 4 592 | 5,5 | 267 |
| FC | 35 582 | 262 | 802 069 | 4 092 | 29 | 123 403 | 7 569 | 4,7 | 105 |
| FC-BJ | 16 839 | 262 | 280 302 | 1 877 | 29 | 330 348 | 7 102 | 2,4 | 39 |
| FC-CBJ | 10 361 | 262 | 119 767 | 1 128 | 29 | 12 247 | 6 506 | 1,6 | 18 |

*Tabulka 1 Výsledky pro výpočty ZEBRA [4]*

V tabulce 1 jsou zobrazeny výsledky ZEBRA problémů. Sloupce s názvem Min obsahují hodnoty pro nejlepší případ, sloupce s názvem Max obsahují hodnoty pro nejhorší případ a sloupce s názvem μ obsahují průměrnou hodnotu ze všech 450 příkladů.

Jak lze vyčíst z tabulky pořadí algoritmů je u každého testu jiné. V prvním testu je pořadí algoritmů: BT – BJ – BM – BMJ – CBJ – FC – BM-CBJ – FC-BJ – FB-CBJ (řazeno od nejhoršího k nejlepšímu algoritmu). Rozdíl v tomto seřazení je lepší pozice backmarkingu, který má stejný prostor jako backtracking, avšak díky jeho funkcionalitě se zmenšil počet kontrol.

V druhém testu je pořadí uvedeno na obrázku 8. Jak lze z tabulky vidět, tak backtracking a backmarking mají stejný prohledaný prostor. Je to tím, že backmarking pouze zabraňuje nadbytečnému testování omezení. Toto také platí pro algoritmy v kombinaci s backmarkingem.

Při posledním testu vypadalo pořadí algoritmu následovně: BM – BT – BJ – BMJ – CBJ – BM-CBJ – FC – FC-BJ – FC-CBJ. Zde byl největší rozdíl oproti předchozím seřazením algoritmus backmarking, který má menší počet kontrol omezení, ale má vysokou cenu k přístupu a úpravě pole elementů obsahující informace, kdy se má provést kontrola a kdy nikoliv.

Bylo dohadováno, že řazení algoritmů by mělo být podle času potřebného k vykonání, než podle velikosti prohledávaného prostoru. Avšak i zde jsou algoritmy seřazeny velmi podobně k dvěma předchozím seřazením. Algoritmy se dají rozdělit do tří skupin: rychlé: FC, FC-BJ, FC, CBJ, CBJ BM-CBJ, kdy tyto algoritmy se vždy umístily na nejlepších pozicích nezávisle na typu řazení. Další skupina by byla skupina průměrných algoritmů: BMJ a BJ a v poslední řadě pomalé algoritmy: BM a BT.

# 6 Technologie

Program je vytvořený jako webová aplikace. Aplikace využívá základní webové technologie, kterými jsou:

HTML - Tato technologie se stará o vzhled stránky, jejím hlavním úkolem je poskytnout uživatelské grafické rozhraní neboli GUI.

CSS - Kaskádové styly jsou v současné době součástí každé webové stránky. Tyto styly slouží k zpřehlednění a stylizování elementů na stránce.

Bootstrap - Knihovna postavená na kaskádových stylech. Obsahuje velké množství předdefinovaných stylů, které zlehčují vytvoření designu.

GoJS[5] - Knihovna, která se stará o vykreslování výsledné stromové struktury. Tato knihovna usnadňuje práci s vykreslením grafu a to především, že v základním HTML chybí jednoduchá komponenta k vykreslení grafu (HTML obsahuje pouze základní canvas, který umožnuje vykreslování, avšak bez použitých knihoven je jeho použití velmi komplikované a zdlouhavé). Navíc tato knihovna obsahuje velké množství pro nastavení jak má graf vypadat a jak se chovat.

Javascript - Tato technologie běží na klientském zařízení a hodí se všude tam, kde by bylo zbytečné používat server. V této aplikaci je využitý pro veškeré výpočty a ovládaní aplikace.

Angular - Tento Framework je postaven na technologii javascript. Jednou z jeho výhod je používání modulů. Využívá svázání elementů a to umožnuje snazší práci se vstupy a výstupy. Angular využívá type-script, což je programovací jazyk postavený na javascriptu, a umožnuje lepší využití tříd a objektů.

PrimeNG - Knihovna jenž obsahuje velké množství komponent, které ulehčují vytváření a práci s grafickým uživatelským rozhraním.

JSON – Technologie umožňující práci se soubory. Používá se především pro import a export vytvořených příkladu a k práci se slovníky.

PHP – Tato technologie se používá pro výpis ukázkových příkladů nacházejících se na serveru a jejich načtení ze serveru.

Jasmine – knihovna určena pro vytváření automatických testů.

# 7 Architektura

Aplikace je postavena na principu tlustého klienta, což znamená, že většina, v této aplikaci celá logika je přesunuta na stranu uživatele a se serverem komunikuje minimálně. Výhoda tohoto klienta je ta, že poté, co si uživatel načte aplikaci ze serveru, může využívat aplikaci i offline.

Dále je aplikace postavena na principu single-page application. Tento princip spočívá v tom, že veškerý běh aplikace se vyskytuje pouze na jedné webové stránce, což dává lepší uživatelský zážitek díky tomu, že se nemusí přecházet z jedné stránky na jinou nebo se stránka znovu nenačítá. K tomu jsou právě nezbytné technologie jako je javascript a angular. Angular kvůli svázání elementů, což pak upravuje data kdykoliv se změní vzhled. Tyto technologie umožňují funkci single-page application tak, že vytvářejí a posílají HTML kód, který prohlížeč následně znovu vyrenderuje jako nový vzhled stránky.

Další principem architektury je třívrstvá architektura. Tato architektura je softwarově rozdělena na tři části, kterými jsou:

Prezenční vrstva - se stará o vzhled stránky. Umožňuje uživateli zadávat požadavky a následně slouží k zobrazení výsledků na dané požadavky.

Aplikační vrstva - se stará především o zpracování požadavků získaných ze vstupů a komunikaci s datovou vrstvou. Poté, co jsou výsledky zpracovány, je zaslaná odpověď na prezenční vrstvu, která tyto výsledky zpracuje a převede na vzhled.

Datová vrstva - slouží k ukládání dat, většinou to bývá do databáze. V této aplikaci je datová vrstva tvořena třídami objektu, které jsou uloženy pouze v paměti, tudíž nejsou trvale uloženy a po znovunačtení stránky již data nejsou k dispozici.

# 8 Funkčnost aplikace

V podkapitole budou popsány jednotlivé vrstvy programu a jejich funkce.

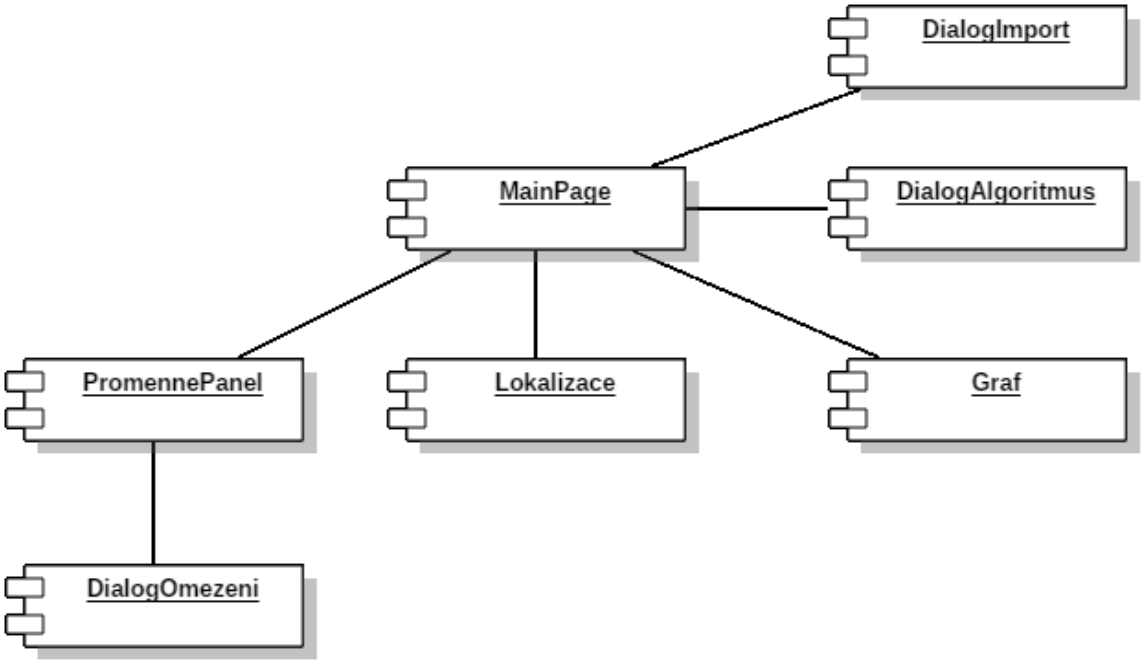
## 8.1 Prezenční vrstva

Po načtení aplikace si uživatel nastaví požadovaný počet proměnných pomocí tlačítka +. Tyto proměnné představují objekty proměnných uložených v datové vrstvě. Těmto proměnným se nastaví jejich domény, což jsou prvky, které se budou proměnné postupně přiřazovat, dokud se po nastavení správných prvků z domén proměnných nenajde řešení. Další vlastností, kterou uživatel nastaví proměnným, je omezení mezi jednotlivými proměnnými, které musí být splněny, aby nedošlo k stavu uváznutí.

Uživatel zadává prvky jako celá čísla oddělena čárkou. Tato čísla se potom validují a uloží se do pole prvků domény. V případě, že číslo z nějakého důvodu neprojde validací (např. nebylo zadáno číslo, ale písmeno), pak se tento prvek smaže z pole zadávající prvky domény. Další akcí je zadávání omezení proměnných. Omezení se zadávají pomocí povolených a zakázaných dvojic. Nejprve se zadá, s jakou proměnnou se mají dvojice porovnávat a dále se již zadají samotné dvojice. Zde opět platí jako pro prvky domén, že jednotlivé části dvojice musí projít validací, jinak je celá dvojice ignorována. Uživatel si může zvolit s kolika proměnnými má povolené nebo zakázané dvojice. Toto je hlavní způsob zadávaní omezení avšak pro zjednodušení je zde možnost zadat, zda přiřazený prvek má být menší, větší, stejný nebo se nesmí rovnat prvku zadané dvojice. Při tomto způsobu uživatel zadává u vybraného omezení proměnné pomocí vepsání názvu proměnné, se kterou se mají prvky domény porovnat. Poté opět dojde k validaci, která zkontroluje, že zadané proměnné skutečné existují. Tento druhý způsob zadávání omezení je zde pro zjednodušení, jelikož tato omezení se dají vytvořit pomocí povolených nebo zakázaných dvojic. Navíc se tyto dva způsoby dají kombinovat.

Poté, co uživatel zadá domény a omezení nastaveným proměnným si vybere algoritmus, který má najít řešení. Jakmile se najde řešení, je vytvořen graf stromové struktury, který si uživatel může nechat vykreslit. Pro možnost vykreslení je zde několik způsobů. Buď si nechat vykreslit jeden krok, vykreslit si najednou deset kroků nebo si nechat vykreslit celý graf najednou. Graf je ve formě stromové struktury a tedy každá řada představuje jednotlivé prvky, jak byly postupně přiřazeny proměnné. V případě oranžového uzlu se jedná o stav uváznutí a nastává zpětná fáze. V případě šedého uzlu se jedná o splnění všech omezení a nastává dopředná fáze. Jestli je uzel zelený, znamená to nalezené řešení a nastává zpětná fáze. Takto se projdou všechna přiřazení.

Prezenční vrstva je vytvořena pomocí různých komponent. Některé komponenty jsou použity z knihovny primeNG další jsou vlastnoručně naprogramované.




*Obr. 9 – Komponenty prezenční vrstvy*

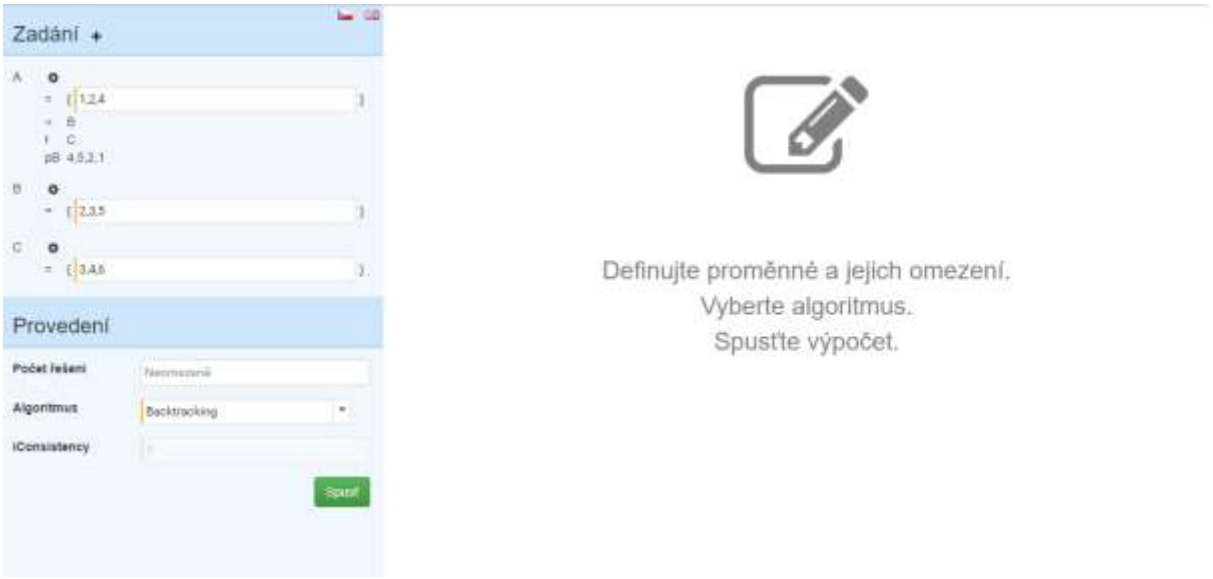
### 8.1.1 Uživatelské rozhraní

Aplikace je zpracována jako webová aplikace běžící v prohlížeči. K vlastnostem tohoto typu programů patří komplexnost a provázanost dat, jejich jednoduché zadávání a ovládání. Požadavek na přehlednost je zajištěn jednoduchým rozhraním, kde se data nacházejí na jednom místě, aniž by se muselo přecházet z jednoho rozhraní na jiné.

Jak lze vidět na obrázku č. 10 tak se tato aplikace skládá z jediné stránky, na které se provádí zadávání vstupních dat, jejich zpracování a také zobrazení výstupních dat. Rozhraní je rozděleno do několika logických částí.

První část slouží k nastavení a výpočet algoritmů. Nastavuje se zde počet proměnných a kolik se má najít řešení před ukončením algoritmu. U proměnných se nastavuje jejich doména, tedy hodnoty, které může proměnná nabývat, pořadí ve kterém se mají proměnné inicializovat a omezení mezi jednotlivými proměnnými. Další možností je vybrat si jeden z předdefinovaných příkladů. Nakonec si uživatel vybere, pomocí jakého algoritmu se problém má zpracovat.

Druhá část potom slouží k ovládání diagramu a jeho vykreslování. Jak je vidět na obrázku č. 10 po načtení stránky se v části grafu zobrazuje informace, jak má uživatel s aplikací pracovat. Poté co si uživatel nechal vypočítat některý algoritmus, změní se informace o aplikaci na oblast, kde si uživatel nechá vykreslit graf. U ovládání grafu existují možnosti nechat si zobrazit celý graf najednou nebo si ho nechávat vykreslovat postupně buď po jednom kroku, nebo vykreslit graf až do dalšího řešení jak dopředu tak zpátky. Dalšími prvky ovládání grafu je možnost při velkém diagramu nechat si diagram zmenšit tak, aby se vešel na celou vytyčenou plochu, a nakonec možnost přiblížení. Pomocí knihovny GoJS je možnost upravit tak, že se může přesunout každý uzel, pomocí skrolování si nechat diagram přiblížit nebo oddálit.



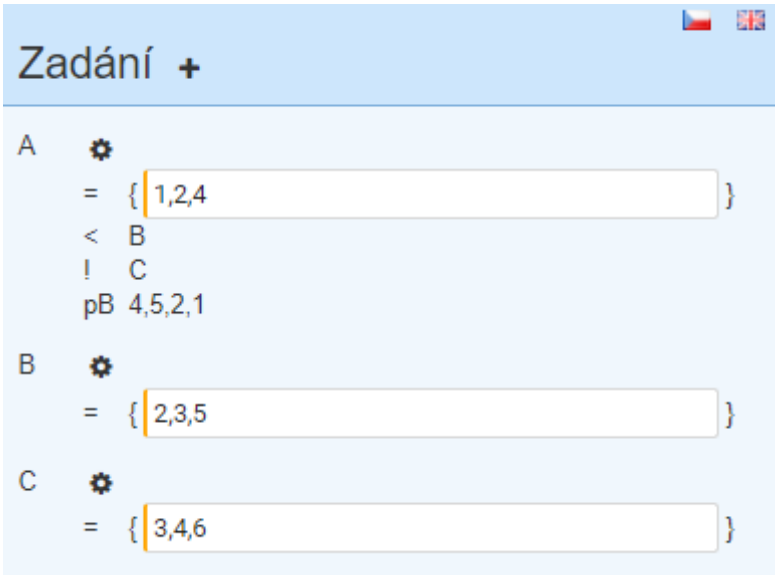

*Obr 10 – Uživatelské rozhraní*

### 8.1.2 Nastavení aplikace

Jak bylo řečeno, uživatel potřebuje nastavit počet proměnných a hodnoty jejich domén. Dále může nastavit požadovaná omezení proměnných.

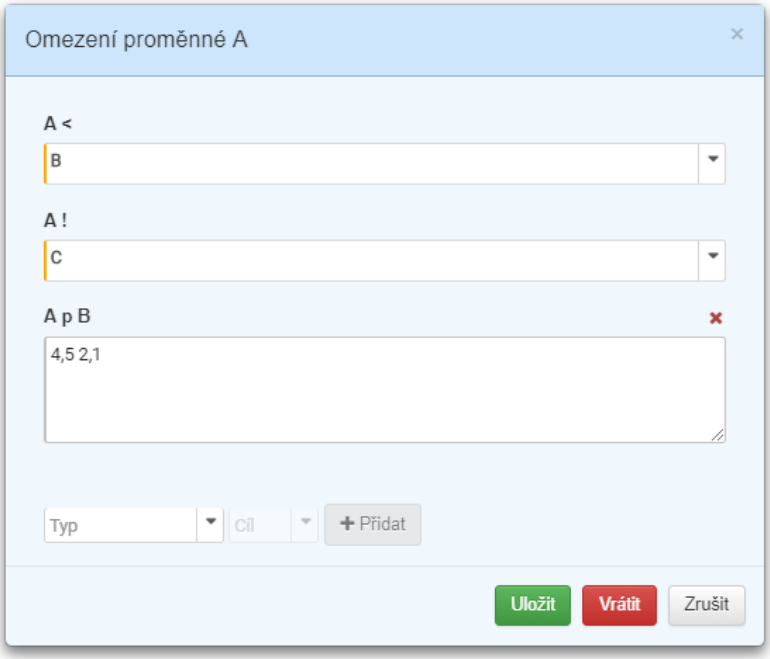
Poté, co se klikne na tlačítko přidat (+), tak je zavolána metoda, která vytvoří objekt nové proměnné a nastaví její doménu a seznam omezení na prázdnou hodnotu a přiřadí jí identifikátor (název), podle kterého lze jednoznačně určit, o kterou proměnnou se jedná. Pro tuto právě vytvořenou proměnnou se v GUI vytvoří elementy, pomocí kterých se proměnné nastaví její další vlastnosti.

Vedle identifikátoru proměnné se nachází vstupní pole, do kterého se zadávají prvky domény. Tyto prvky se zadávají pomoci přirozených čísel. Nad tímto vstupem se provádí kontrola a jiný znak než přirozené číslo není možné přiřadit do domény. Je požadováno, aby doména obsahovala aspoň jeden prvek, jelikož pro prázdnou doménu neexistuje řešení.




*Obr 11 – Ukázka proměnný, jejich domén a omezení*

Další vlastností nastavitelnou pro proměnnou jsou omezení. Po kliknutí na tlačítko nastavení omezení (☼) se otevře modální okno, kde bude možné nastavit omezení. Uživatel má na výběr buď základní omezení, kde po jeho výběru může přidat toto omezení a poté, co je toto omezení vytvořeno, vybere mezi kterou proměnnou se toto omezení musí splnit. Je možno vybrat několik proměnných pro stejný typ omezení avšak je požadováno, aby byla přiřazena aspoň jedna proměnná. V případě, že typ omezení je povolené nebo zakázané dvojice, musí před vytvořením omezení být ještě uvedeno, se kterou proměnnou se mají dvojice porovnávat. Poté, co je omezení vytvořeno, uživatel zadává dvojice. Dvojice jsou opět přirozená čísla.




*Obr 12 – Okno pro zadávání omezení*

Dále je zde možnost zakázat proměnnou. Pokud je proměnná zakázaná, tak se s touto proměnnou při algoritmu nepočítá, avšak zůstává uložena a je možnost ji opět aktivovat. Poslední možností je proměnnou smazat ze seznamu.

Po nastavení proměnné si uživatel může zvolit jeden z algoritmů, pomocí kterého se hledá vyhledá řešení. Před spočtením algoritmu si uživatel může zvolit, po kolika nalezených řešeních se má algoritmus ukončit (defaultně nastaveno na nalezení všech řešení). V případě algoritmu i-consistency si může nastavit hodnotu *i*, která reprezentuje počet potřebných přiřazených proměnných potřebných pro ponechání prvku v doméně. Nakonec po výpočtu algoritmu si uživatel může postup algoritmu nechat vykreslit pomocí grafu.

### 8.1.3 Popis algoritmu

Uživatel si vybírá, jakým algoritmem se má najít řešení požadovaného algoritmu, avšak nemusí vědět, jak daný algoritmus přesně funguje. Nachází se zde možnost zobrazit si informace o funkčnosti algoritmu po kliknutí na tlačítko ? nacházející se vedle vybraného algoritmu. Po kliknutí na tlačítko se otevře modální okno, které obsahuje detailní popis toho, jak algoritmus funguje.

### 8.1.4 Internacionalizace

Jelikož aplikace je určena pro studenty univerzity, lze předpokládat, že ji budou využívat i studenti ne hovořící česky. Z tohoto důvodu je v aplikaci zakomponována internacionalizace, jež umožňuje přepínat mezi jazyky, a to českým a anglickým jazykem. Český jazyk se používá z důvodu, že největší část studentů je česky hovořících. Anglický jazyk byl zvolen z důvodu, že je na Vysoké škole Báňské velmi často používán při výuce. Jedna ve zdrojových materiálech tak někdy i při výkladech.

Změna jazyku probíhá dynamicky, to znamená, že změna se provede bez nutnosti znovu načíst stránku a je tedy možnost přepnutí, aniž by to přerušilo práci s aplikací. Internacionalizace funguje na principu slovníků, kdy je pro každý jazyk vytvořen slovník obsahující texty v tomto jazyce. Po načtení aplikace se na začátku načte výchozí jazyk, potom se po přepnutí jazyku načte požadovaný slovník. V kódu programu se nenacházejí přímo texty, ale adresa odkazující se na místo ve slovníku, kde se daný text nachází.

Například při zadávání domény proměnné je nutné splňovat některé požadavky, a pokud tyto požadavky nejsou splněny, tak se vypíše chybová hláška. V kódu je odkaz na adresu pomocí `{{'promenna.validace.pattern' | translate}}`. Toto je odkaz na adresu ve slovníku. Zde je výpis slovníku pro danou adresu pro českou verzi:

```
„promenna“: {
      „header“: „Zadání“,
      „validace“: {
            „pattern“: „Povolena jsou celá čísla oddělená čárkou“
      }
```

*Kód 1 – ukázka obsahu slovníku, česká verze*

A zde se nachází slovník pro anglickou verzi:

```
„promenna“: {
„header“: „Inputs“,
„validace“: {
    „pattern“: „Only integers are allowed with comma separator“
}
```

*Kód 2 – ukázka obsahu slovníku, anglická verze*

Jak je vidět ze slovníku a adresy, tak adresa *promenna* má další dvě podadresy *header* a validace, navíc adresa *promenna.validace* má podadresu *pattern*, kde se nachází daná zpráva, která se na požadovaném místě vypíše. Do slovníku se také dají zapisovat údaje z rozličných proměnných pomocí parametru, který je ve slovníku znázorněn pomocí dvojí složených závorek např.: "Vybírám první volnou hodnotu z domény proměnné *{{nazev}} = {{hodnota}}*."

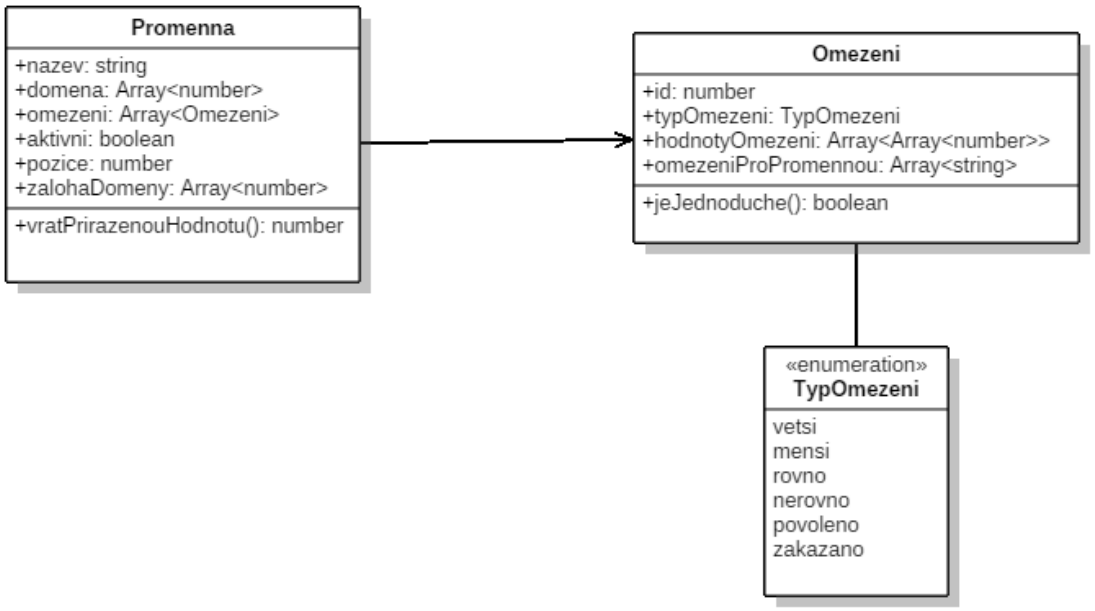
## 8.2 Datová vrstva

Tato vrstva slouží pro práci s daty, a to především k jejich ukládání a následovnému získáváním. Většinou se data ukládají do databáze, avšak v této aplikaci se data ukládají pouze in memory, tedy data jsou uložena pouze do proměnných a po použití jsou zapomenuta. Je to z důvodu, že v této aplikaci se jedná o jednoduchá data, která se okamžitě zpracují. Po zpracování data stále zůstávají uložena v paměti. Pokud si uživatel přeje data uchovat pro pozdější opětovné použití, je možné tato data vyexportovat do souboru, a poté je uživatel může kdykoliv znovu naimportovat. Tato data je možné také uložit na server, kde budou k dispozici všem uživatelům kdykoliv a kdekoliv, avšak tyto soubory na server může uložit pouze správce.

Datová vrstva je tvořena pomocí několika objektů. Prvním takovým objektem je *Promenna*. Tento prvek je velmi důležitý, jelikož uchovává všechny informace, které se vztahují k proměnné. Nejdůležitějšími vlastnostmi tohoto objektu je doména a omezení. Doména obsahuje prvky, které se budou postupně přiřazovat. Tyto prvky se zadávají v GUI aplikace v sekci zadání. Omezení je další typ, který se probere níže. Dalšími vlastnostmi je název proměnné, informace o tom, jaké prvky se mají ještě přiřadit, než začne zpětná fáze. Nakonec obsahuje informaci o tom, jestli je daná proměnná aktivní. V této aplikaci si uživatel může zvolit deaktivovat proměnnou, aby se s touto proměnnou ve výpočtech nepočítalo, avšak uživatel tuto proměnnou nechce smazat, jelikož by ji chtěl znovu použít. V tomto případě stačí proměnnou pouze aktivovat a již nemusí znovu vypisovat doménu a její omezení.

Dalším objektem datové vrstvy je objekt Omezeni. Tento objekt obsahuje především informaci, o jaký typ omezení se jedná. Všechny typy omezení jsou uloženy v enumerátoru. Další informace je s jakou proměnnou se omezení musí splnit a obsahuje pole proměnných. V případě, že omezení je typem povolené nebo zakázané dvojice obsahuje pouze jednu proměnnou, s jakou se musí splnit omezení,

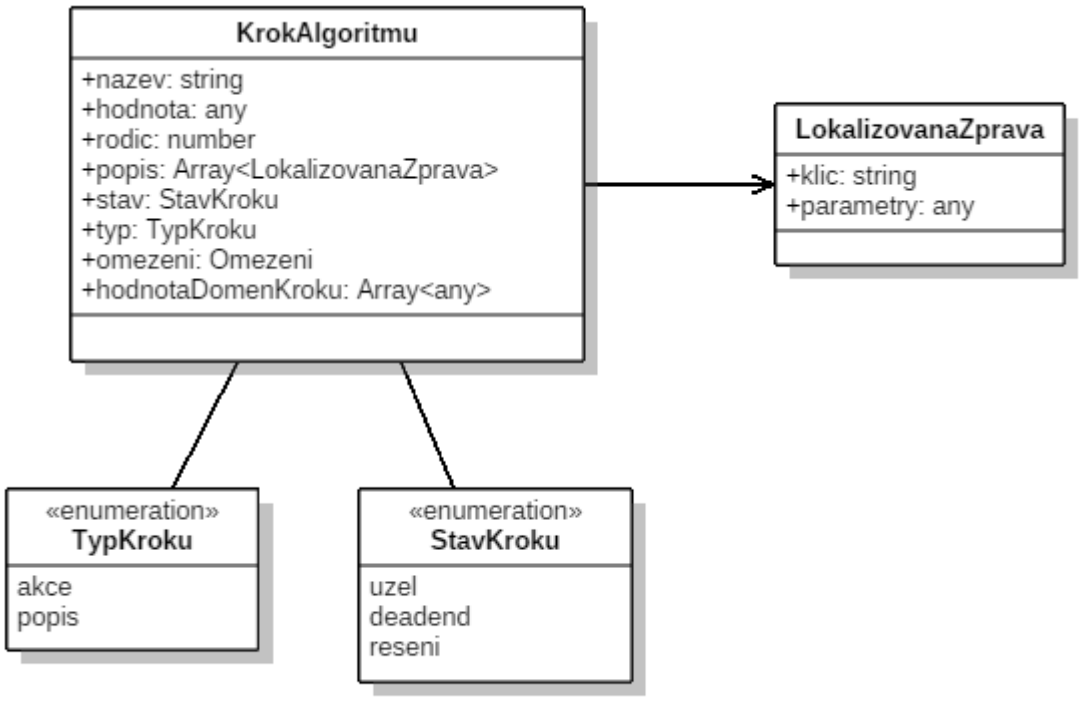
avšak navíc obsahuje seznam povolených dvojic.




*Obr. 13 – Třídní diagram datové vrstvy pro proměnné*

Dalším objektem této vrstvy je *KrokAlgoritmu*. Tento objekt se stará o informace popisující co se událo v každém kroku algoritmu. Obsahuje identifikaci proměnné, dále hodnotu, která byla pro proměnnou v prováděném kroku přiřazena. Dalším velmi důležitým prvkem je informace o předkovi. Tato informace slouží v diagramu k tomu, aby se zjistilo, který uzel je předek tohoto uzlu. Následující informace obsahuje stav uzlu, může být jako *uzel*, *deadend* a *reseni*. Podle toho, jaký stav je přiřazen, se nastaví barva pozadí uzlu. Pro stav *uzel* to bude šedá barva, stav *deadend* oranžová a stav *reseni* bude mít barvu zelenou. Další informaci je hodnota domén jednotlivých proměnných v daném kroku. Tato informace se využívá pouze v některých algoritmech, a to především v těch, v kterých se upravují domény proměnných. Poslední informace je popis, co se v tomto kroku provedlo, která se vypíše pro uživatele. Tento popis je datového typu *LokalizovanaZprava*.

Posledním objektem je *LokalizovanaZprava.* Jak již bylo řečeno, tak je tato aplikace internacializována. Proto, aby uživatel nemusel znovu vyhledat řešení kvůli změně jazyka obsahuje *LokalizovanaZpráva* dva parametry. Prvním parametrem je klíč, který slouží jako informace o adrese ve slovníku. Druhým parametrem objekt obsahující dvojice: název parametru, který je udán ve slovníku a jeho hodnota.




*Obr 14 – Třídní diagram pro krok algoritmu*

Toto jsou objekty uložené v paměti programu. Data uložena v souborech jsou informace, kterými se naplní objekty Promenna a Omezeni. Tyto informace jsou zpracovány pomocí technologie JSON. Této technologii odpovídá také struktura souboru. Struktura takového JSON obsahuje seznam proměnných uložených v hranatých závorkách, kde každá proměnná je objekt uložený v složených závorkách. Objekt obsahuje dvojici: název proměnné v uvozovkách a jakou hodnotu tato proměnná obsahuje. Na konci každé takové dvojice, případně objektu je nutné dát čárku až na poslední objekt. Struktura takových objektů je stejná jako výše popsané objekty *Promenna* a *Omezeni*.

```
[  {
   "nazev": "A",
   "domena": [ 1,2, 4  ],
   "omezeni": [
    {"id": 1,
     "typOmezeni": "!",
     "hodnotyOmezeni": ["C","D" ],
     "omezeniProPromennou": null },
    {"id": 2,
     "typOmezeni": "p",
     "hodnotyOmezeni": [ [ 4, 5 ], [ 2,1 ] ],
     "omezeniProPromennou": "B"
```

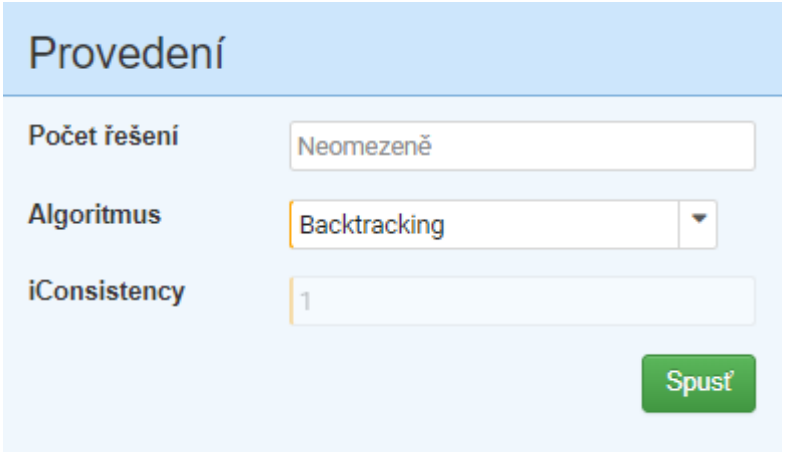
```
      }
    ],
    "aktivni": true
  },
]
```

*Kód 2 – Ukázka souboru JSON obsahující ukázkový příklad*

## 8.3 Aplikační vrstva

Tato vrstva se stará o zpracování požadavků, což znamená zpracování vstupů a výpočet algoritmů. Jednou z prvních věcí, které tato vrstva řeší, je validace vstupu a omezení. Díky svázaní prvku s elementy vstupu, jsou data okamžitě ukládána do proměnných. Jakmile uživatel zašle požadavek, dojde nejprve k upravení omezení. Upravení omezení se provádí proto, že uživatel ne vždy zadá omezení správně, nebo v případě několika algoritmů dojde ke změně pořadí proměnných a omezení je třeba přiřadit správným proměnným. Omezení se přidá do vlastností proměnné z této dvojice, u které dojde až k pozdějšímu přiřazení prvku z domény. To se provede z toho důvodu, že nelze porovnat, zda je splněno omezení, pokud jedna nebo obě proměnné ještě nemají přiřazený prvek z domény. V případě povolených a zakázaných dvojic, se musí v případě přeřazení omezení také prohodit hodnoty u každé dvojice, jelikož první číslo z této dvojice se ověřuje oproti právě vybrané proměnné a druhé číslo této dvojice se porovnává s proměnnou, která je nastavena v tomto omezení. V případě ulehčujících omezení se musí omezení přiřadit k správnému typu, například pokud je typ omezení menší než a pokud se omezení převede k druhé proměnné, tak se typ omezení musí změnit na větší než.

Jakmile jsou provedeny všechny validace a převody omezení tak jako další krok se provede výpočet algoritmu, který si uživatel zvolil.




*Obr 15 – Výběr algoritmu a počtu požadovaných řešení*

### 8.3.1 Ukázkové příklady

Pokud uživatel nechce zadat vlastní příklad, může využít některý z předpřipravených příkladů, které jsou nasazeny spolu s programem na serveru. Podle načteného ukázkového příkladu aplikace sama nastaví počet proměnných a jejich domény a také omezení, která se musí splnit. Tyto příklady jsou ve formátu JSON a správce programu si může tyto ukázkové příklady sám vytvořit a nasadit je na server, kde se jako název ukázkového příkladu použije název souboru.

Další možnosti, jež aplikace poskytuje, je import a export příkladů. Poté, co si uživatel nastaví počty proměnných, naplní jejich domény a nastavení omezení požadovaným proměnným, tak si uživatel může zvolit vyexportovat zadaný příklad. Příklad bude vyexportován jako JSON soubor a poté si jej uživatel, který má takový soubor k dispozici, může kdekoliv naimportovat a aplikace se podle tohoto souboru nastaví. Takový soubor je možno sdílet a v případě správce serveru tento příklad nahrát na server, kde bude veřejně k dispozici mezi ukázkovými příklady.

# 9 Implementace

## 9.1 Backtracking

Tento základní algoritmus využívá více  algoritmů, přičemž některé pouze upravují některou jeho činnost. Algoritmus se dělí na tři části.

Kontrolní fáze je zpracování proměnné. Na začátku se zkontroluje zda doména obsahuje ještě některé prvky, které nebyly vybrány. Jestli ještě obsahuje prvky tak se vezme první z těchto prvků a přiřadí ho proměnné. Poté dojde ke kontrole, jestli jsou pro daný prvek splněna všechna omezení, která má tato proměnná přiřazena. Jestli projde, nastává dopředná fáze. V opačném případě dojde ke zpětné fázi.

Dopředná fáze. Při této fázi se provede kontrola, zda ještě existuje nepřiřazená proměnna. Pokud ještě existuje nepřiřazená proměnná, tak se nastaví následující proměnná za vybranou a opět proběhne kontrolní fáze. Pokud již neexistuje proměnná, která ještě nemá přiřazen prvek z domény, znamená to, že bylo nalezeno řešení problému, což se zaznamená a začne zpětná fáze.

Zpětná fáze. Tato fáze nastane v případech, kdy se má proměnné přiřadit prvek z domény a ta již neobsahuje žádný prvek, který nebyl proměnné přiřazen. Další z důvodů, kdy nastane tato fáze, je ten, že není splněno aspoň jedno omezení pro daný prvek, který je zrovna přiřazen k vybrané proměnné. Nakonec tato fáze začne pokud je nalezeno řešení. V případě, že doména proměnné ještě obsahuje nepřiřazené prvky, tak se spustí první fáze, která přiřadí následující nepřiřazený prvek. V případě, že doména již neobsahuje prvek, který by se mohl proměnné přiřadit, tak se všechny prvky z domény označí za nepřiřazené a přejde se na předchozí proměnnou, u které se opět spustí první fáze s následujícím prvkem.

Jakmile jsou zkontrolovány všechny možné kombinace prvků u všech proměnných, tak algoritmus skončil.

```
while (nenalezeno počet požadovaných řešení) {
   zpracovavanaPromenna následující pozice;
   if(zpracovavanaPromenna pozice je mimo doménu) {
      nastav zpracovavanaPromenna pozici na neinicializovanou;
      zpracovavanaPromenna je předchozí proměnná v pořadí;
      continue;
      }

   Vytvoření krokuAlgoritmu a uložením do postupuAlgoritmu;
   if (nesplnění omezení) {
     krokAlgoritmu je uváznutí
      } else {
        if (proměnna je poslední inicializována proměnná) {
            zvyš počet nalezených řešení;
           krokAlgoritmu je řešení;
            } else {
```

```
                zpracovavanaPromenna je další proměnná v pořadí;
              }
          }
      }
 return postupTvoreniGrafu;
```

*Kód 3 Pseudokód pro algoritmus backtracking*

## 9.2 Backjumping

Tento algoritmus se velmi podobá backtrackingu. První a dopředná fáze jsou u obou algoritmů identické, liší se pouze ve zpětné fázi. V této fázi se v případě návratu na předchozí proměnnou nejde na proměnnou předcházející, ale zkontrolují se všechna omezení, aby se zjistilo, na jakou proměnnou se může provést skok, aniž by nedošlo k přeskočení nějakého možného řešení. Tato proměnná se zjistí tak, že se získají proměnné, které mají s vybranou proměnnou nějaké omezení a z těchto proměnných se vybere tak, která byla přiřazena naposled. Poté, co se zjistí proměnná, tak se provede skok na tuto proměnnou. Dále se všem proměnným, které byly přiřazeny pro tyto proměnné, odeberou jejich přiřazené prvky a všechny prvky z jejich domén jsou nastaveny jako nepřiřazené. Nakonec začne opět první fáze.

```
if (zpracovavanaPromenna je na konci domény) {
  backjump = první proměnná
  if (zpracovavanaPromenna není leafend) {
    zpracovavanaPromenna nastavení leafendu;
    skok na předchozí proměnnou
    }
    if (zpracovavanaPromenna nemá omezení) {
       skok na předchozí proměnnou
       } else {
        for (počet omezení zpracovavanaPromenna) {
          for (počet hodnot konkretního omezení) {
           if (porovnávaná proměnná u omezení je větší než proměnná
pro skok) {
                  backjump  = porovnávaná proměnná;
                  }
              }
          }
     }
     for (od zpracovávané proměnné do proměnné pro skok) {
         nastavení pozice v doméně na neinicialozováno
         nastavení, že se jedná o leafend
         }
         if (krok není řešení) {
             nastav krok na uváznutí;
             }
```

```
        skok zpět na backjump;
        }
```

*Kód 4 – Pseudokód pro algoritmus backjumping*

## 9.3 Consistency algoritmy

Tyto algoritmy využívají algoritmus backtracking bez jeho změny, avšak liší se v přípravě před samotným backtrackingem. Na rozdíl od backtrackingu se provede příprava před jeho spuštěním, která se pokusí zmenšit domény všem proměnným. V případě Arc-consistency se budou postupně vybírat prvky z domény a provede se kontrola zda existuje aspoň jeden prvek, který splňuje omezení. Pokud žádný takový prvek neexistuje, tak se vybraný prvek odstraní z domény. Path comsistency funguje na stejném principu, avšak se musí najít aspoň dva prvky, které splňují omezení, aby nedošlo k odstranění prvku z domény. Totéž platí pro algoritmus i-consistency kde počet prvků, které musí splňovat omezení je roven i. V případě, že i je větší než je počet omezení přiřazených k této proměnné, musí být splněna všechna omezení. V případě, že i je rovno nule nebo menšímu číslu, tak se tato část přeskakuje a začne výpočet pomocí algoritmu backtrackingu.

Algoritmy popsané výše se vykonaly pomocí algarotimu iConsistency. Avšak nachází se zde ještě upravený algoritmus arc-consistency, který je vykonější než klasický arc-consistency. Obyčejně arc-consistency kontroluje, zda pro daný prvek existuje předchozí prvek, který splňuje omezení. Tento upravený algoritmus však navíc kontroluje omezení i druhým směrem. Tedy kontroluje navíc, zda existuje nějaký prvek pro následující proměnnou, se kterou má omezení. To znamená, že prvky se odstraňují z obou proměnných mezi kterými jsou omezení.

```
for (všechny proměnné) {
 nastala změna = false;
 for (omezení zpracovávané proměnné) {
  switch (typ omezení) {
   case "typ":
    for (obsah omezení) {
     kontrola = splňujeOmezení(porovnavanaPromenna,
zpracovavanaPromenna);
      if (kontrola selhala) {
       ukončení algoritmu s prázdnou doménou u zpracovavanaPromenna
      }
      kontrola = splňujeOmezení(zpracovavanaPromenna,
porovnavanaPromenna);
      if (kontrola selhala) {
       ukončení algoritmu s prázdnou doménou u porovnavanaPromenna
      }
    }
   break;
   if (nastala změna) {
     projít od začátku;
```

*Kód 5 – Pseudokód pro algoritmus arc-consistency*

## 9.4 Forward checking

Tento algoritmus funguje na podobném algoritmu jako consistency, kdy se odstraňují prvky z domén, které způsobí stav uváznutí, avšak toto probíhá během algoritmu. Jakmile začne dopředná fáze, tak se nejprve provede fáze zmenšení domén. Při této fázi se zkontrolují všechny následující proměnné a odstraní se z jejich domén všechny prvky, které nesplňují omezení s právě přiřazeným prvkem. Poté, co se zkontrolovaly všechny prvky všech proměnných, se pokračuje v dopředné fázi jako u backtrackingu. Další rozdíl je ve zpětné fázi algoritmu. Jelikož jsou odstraněny všechny prvky tak zpětná fáze.

```
Provedení zálohy
 for (proměnné následující v pořadí po zpracovavanaPromenna) {
  for (pro všechna omezení porovnávané proměnné) {
   switch (typOmezeni) {
    case "typ":
    for (domenu porovnávané proměnné) {
     if (hodnota nesplňuje omezení) {
      odstranění hodnoty
      if (doména porovnávané proměnné === 0) {
       obnov domény ze zálohy;
       return uváznutí;
      }
     }
    }
    break;
   }
   }
  }
 }
}
Return pokračování v algoritmus upravenými doménami
```

*Kód 6 - Pseudokód pro provedení forward check*

```
for (proměnné nasledující zpracovavanaPromenna) {
 if (proměnná má zálohy) {
  if (záloha se provedla pro při FC zpracovavanaPromenna) {
  obnov zálohu
  }
 }
}
```

*Kód 7 - Pseudokód pro obnovení zálohy u forward check*

Zpětná fáze začne buď tím, že je nalezeno řešení nebo když se v dopředné fázi u některé proměnné odstraní všechny prvky z její domény, což znamená, že pro současné přiřazení neexistuje žádné řešení. Jakmile dojde k návratu na předchozí proměnnou, tak se zároveň musí vrátit všechny prvky, které byly z domén odstraněny pro vybraný prvek. První fáze probíhá podobně jako při backtrackingu, avšak rozdíl u tohoto algoritmu spočívá v tom, že se nemusí provádět kontrola omezení, jelikož doména proměnné obsahuje pouze prvky, které splňují všechna omezení.

## 9.5 Backtracing s náhodným výběrem

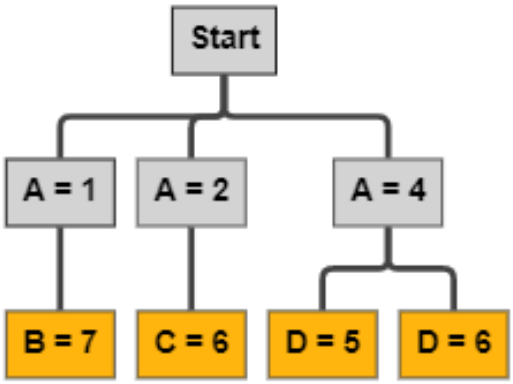
Tento algoritmus funguje na stejném principu jako backtracking, jediná změna je v okamžiku přiřazení prvku z domény. U backtrackingu se prvky z domény přiřazují postupně, jak jsou v doméně uloženy, avšak u náhodného výběru se proměnné přiřadí úplně náhodný prvek. Poté je prvek označen jako již přiřazený, z toho důvodu, aby nebyl opět přiřazen.

## 9.6 Dynamic Value Ordering

Tento algoritmus funguje jako obyčejný backtracking, avšak před jeho spuštěním se změní proměnným pořadí. Toto pořadí je určeno podle velikosti domény proměnných, kdy na první místa přijdou proměnné s nejmenší doménou a na konec proměnné s největší doménou. Nedojde ke zmenšení stromové struktury, jelikož se v obou případech projdou všechny kombinace, ale dojde ke zrychlení díky zmenšení počtu dopředných a zpětných fází.

## 9.7 Forward checking s Dynamic Value Ordering

Toto spojení dvou algoritmů ukáže lepší výhody algoritmu dynamic value ordering. U dynamic value ordering se změní pořadí podle velikosti domény. Jelikož ale dále algoritmus funguje jako obyčejný backtracking, který nemění domény tak se toto uspořádání provede pouze před vykonáním algoritmu. K tomuto účelu je zde algoritmus forward checking, který mění domény všech následujících proměnných po každém přiřazení. Díku těmto úpravám domén může vypadat pořadí jako na obrázku 16. Na tomto obrázku lze vidět, že po proměnná A je následující proměnná B, pro přiřazení dalšího prvku je to proměnná C a pro další prvek je to proměnná D.




*Obr 16 – Ukázka dynamické změny pořadí proměnných při forward checkingu*

## 9.8 Provedení kontroly

Je Jednou z nejdůležitějších částí programu. Tato část se provádí po každém přiřazení prvku proměnné, v případě některých algoritmů se tato část provádí také před samotným algoritmem hledajícím řešení. Na výsledku této části závisí, zda se provede dopředná nebo zpětná fáze. Nejprve se zkontroluje, zda právě zpracovávaná proměnná musí splňovat nějaká omezení. Jestli tato proměnná nemusí splňovat žádné omezení, tak se provede dopředná fáze. V případě, že proměnná musí splňovat omezení, tak se přejde k jejich kontrole.

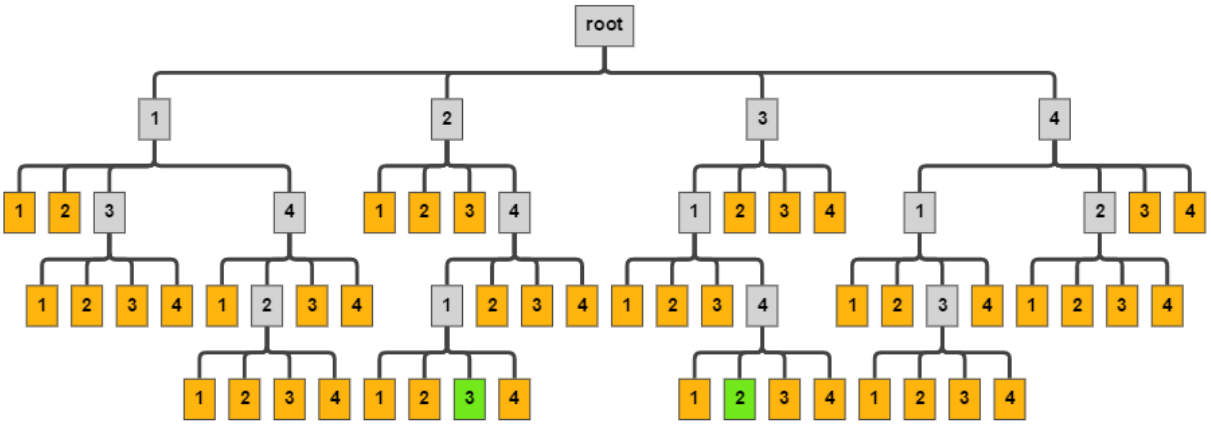
Při kontrole se postupně procházejí všechna omezení. Nejprve se zjistí o jaké omezení se jedná (jestli se jedná o základní omezení nebo povolené či zakázané dvojice). Poté se zjistí s jakou proměnnou se toto omezení musí splnit. Nakonec se zjistí přiřazený prvek této proměnné a provede se kontrola omezení podle jeho typu. Pokud je omezení splněno, tak se přejde k dalšímu omezení. Jakmile jsou všechna omezení zkontrolována a splněna, tak se kontrola ukončí a algoritmus může přejít do dopředné fáze. Jestli dojde při kontrole některého omezení k nesplnění jeho podmínky, tak se kontrola okamžitě ukončí a algoritmus přejde do kontrolní fáze.

## 9.9 Instrukce pro vytváření grafu

Graf je vykreslován pomocí knihovny goJS. Proto aby mohla vykreslit stromovou strukturu probíhajícího algoritmu, potřebuje seznam instrukcí jak tuto strukturu vykreslit. Instrukce jsou zaznamenávány během běhu algoritmu. Každá instrukce obsahuje co se v daném kroku provedlo. Mezi tyto informace patří identifikátor proměnné, se kterou se v daním kroku pracovalo a jaká hodnota se jí přiřadila. Dále rodič, který odkazuje na předchozí přiřazenou proměnnou. Tento rodič identifikuje, na který uzel se má připojit hrana. Stav v jakém se daný krok nachází, tyto stavy jsou průchozí stav, stav uváznutí a nalezení řešení. Popis, jenž obsahuje stručný popis co se v tomto kroku událo. Tento popis se zobrazí po vykreslení daného kroku. Nakonec je zde informace jestli daný krok je aktivní nebo pouze popis.

Aktivní krok znamená, že se provedlo nové přiřazení a toto přiřazení je vykresleno do grafu pomocí uzlu. Aktivní krok také obsahuje informace co se v daném kroku provedlo. Popisný krok značí, že se provedly nějaké akce, které se zaznamenají do popisu, ale nedošlo k žádnému přiřazení nebo nějaké změně, kvůli které by se musel překreslovat graf. Obvykle je mezi aktivními kroky několik popisných kroku, avšak můžou být i aktivní kroky ihned po sobě.

## 9.10 Vykreslování grafu




*Obr. 17 Ukázka vykresleného grafu*

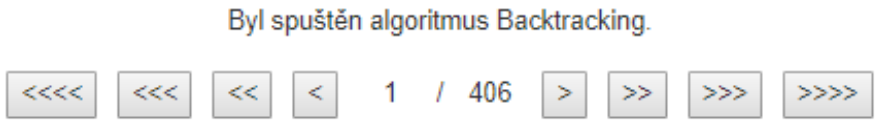
Instrukce, jak se má graf vykreslit, byly zaznamenány při vyhledávacích algoritmech. Poté, co se ukončí vyhledávací algoritmus, se tyto instrukce předají rozhraní knihovny, která se stará o vykreslování diagramu. Nejprve se inicializuje prvek, který se stará o zobrazování diagramu, a jeho funkčnost. Toto se vytváří a nastavuje již při načtení stránky. Při tvorbě této komponenty se zavolá knihovna, po které se bude požadovat komponenta diagram. Této komponentě se nastaví požadované vlastnosti. Mezi tyto vlastnosti patří například rychlost zobrazení nápovědy po najetí na komponentu, dále nastavení vycentrování diagramu na vrchní část vykreslovacího prvku, zakáže se možnost dělat kopie komponent. Důležité je potom nastavení jak se má diagram vykreslovat. Toto je zde nastaveno jako vykreslování pomoci stromové struktury, u které se ještě nastaví úhel, mezery a další prvky.

Pak se nastaví informace o tom, jak mají vypadat komponenty zobrazující uzly. U tohoto nastavení se přidá podružné nastavení o zobrazeném okně nápovědy po najetí myši na uzel. Dalším nastavením u uzlu je vytvoření funkce, která bude kontrolovat, jestli je daný uzel jako normální uzel, uváznutý uzel nebo uzel, u kterého se našlo řešení. Podle tohoto stavu uzlu se nastaví barva pozadí, které se má vykreslit.

Dalším nastavením je nastavení linek spojující jednotlivé uzly. Zde se nastavují základní prvky jako je typ a tvar linky, velikost úhlu a další vlastnosti.

Na závěr zbývá nastavit *event listenery* pro jednotlivá tlačítka a nastavit, jaká funkce se má po stisknutí na dané tlačítko provést.

Po vypočtení algoritmu a předání postupu pro tvorbu diagramu se prvek zobrazující komponentu vymaže pro přípravu na zobrazování nového diagramu. Poté si uživatel zvolí pomocí tlačítek, jakou akci má komponenta nechat zobrazit (obrázek 17).




*Obr. 18 Ovládaní grafu s popisem prováděného kroku*

Akce pro vykreslování grafu (obrázek 18) jsou vždy ve dvojici, pro každou akci existuje stejná akce v opačném směru. První akce se vyvolá po kliknutí na tlačítko >. Tato akce posune graf pouze o jeden krok. Jak již bylo napsáno u datové vrstvy kroky se dělí na dva stavy. V případě, že následujícím krokem bude typu popis, tak se graf nezmění, změní se pouze popis. Pokud bude typ popisu akce vykreslí se nový uzel a vypíše se popis kroku. V Případě opačné akce se graf posune o jeden krok zpět.

Další akce se vyvolá po stisknutí tlačítka >>. Tato akce značí posunutí na další uzel. Na rozdíl od předchozí akce, tato akce přeskočí všechny kroky typu popis a vykreslí do grafu nový uzel. Na tento uzel se vypíše pro uživatele, s jakou proměnnou se v tomto kroku pracovala a jaká ji byla přiřazena hodnota. Dále se uzlu nastaví informace o rodiči, aby se mohl k němu připojit hranou. Nakonec se vypíše popis co se v tomto kroku událo. Opačná akce se vrátí zpět na předchozí uzel do kroku kdy se této proměnné přiřadila nová hodnota.

Následující akce se vyvolá pomocí tlačítka >>>. Po vyvolání této akce se vykreslí graf do dalšího řešení. Přeskočí se tedy všechny kroky, kdy typ kroku bude popis a všechny kroky typu akce, které nebudou mít jako stav nastaveno řešení. Poté co se tento krok najde vykreslí všechny uzly, které se vytvořily do tohoto kroku. V případě, že již neexistuje další řešení, nebo problém řešení nemá, provede se skok až na poslední krok. Opačná akce se vrátí na předchozí řešení. Pokud ještě nebylo nalezeno řešení, na které by se graf mohl vrátit, tak se vrátí na první krok.

Poslední akce se vyvolá pomocí tlačítka >>>>. Tato akce vykresli celý graf najednou do úplně posledního kroku včetně kroku typu popis. Opačná akce potom uvede graf do prvního kroku, ještě před přiřazením prvního prvku první proměnné.

*Obr. 19 Ovládání přiblížení grafu*

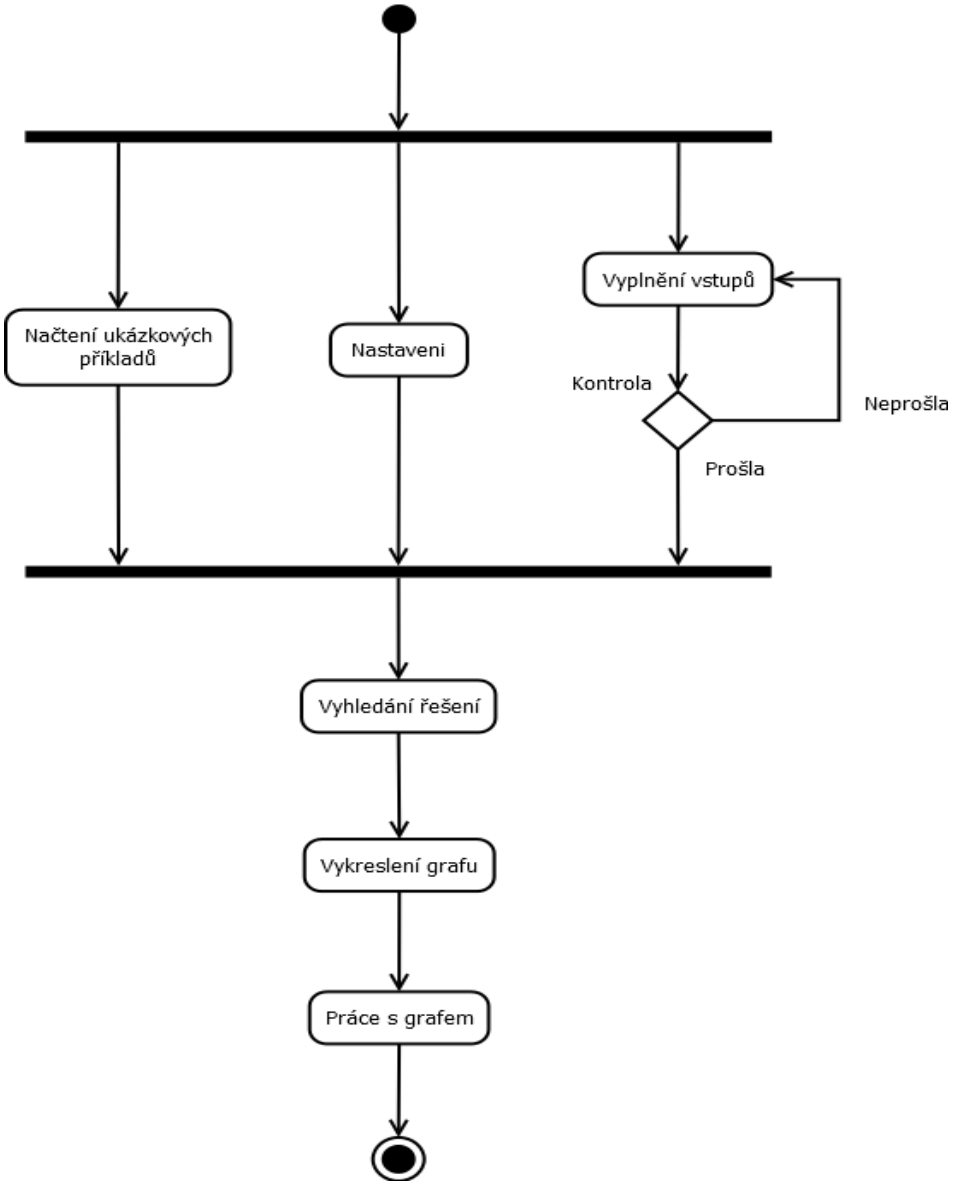
Další akcí jsou přiblížení, oddálení a vycentrování (obrázek 19). Akce přiblížení se vyvolá po stisknutí tlačítka + a dojde k přiblížení(zvětšení) grafu. Tato akce se dá také vyvolat pomocí přidržení klávesy ctrl a skrolováním kolečkem myši nahoru. Akce oddálení se vyvolá po stisknutí tlačítka – a dojde k oddálení(zmenšení) grafu. Tato akce se opět dá vyvolat přidržením klávesy ctrů a skrolováním

kolečkem myši dolů. Poslední akce se provede po stisknutí tlačítka terče. Tato akce nastaví velikost grafu, aby se přesně vešla do dané komponenty avšak pouze v případě, že se do této komponenty v současném přiblížení nevejde. Uživatel může po kliknutí myši a přidržení tlačítka pohybovat s grafem. V případě, že graf se nevejde do dané komponenty uživatel bude muset provést oddálení, nebo skrolovat v grafu.

V případech, kdy se bude zadaný problém vypočítávat pomocí algoritmů forward-checkingu nebo Arc-consistency, tak dojde ke změnám v doménách proměnných. U arc-consistency dojde k zjednodušení domén proměnných před použitím algoritmů backtrackingu a u forward checkingu se budou domény měnit při každém novém přiřazení hodnoty z domény. Proto pokud se použijí tyto algoritmy, tak se vytvoří nad grafem pole obsahující hodnoty v doménách. Pro arc-consistency se po vytvoření těchto polí nahrají obsahy domén, toto bude stačit, jelikož domény se dále v algoritmu již nemění. Pro algoritmus forward-checkingu se do vytvořených polí bude po každém kroku nahrávat obsah domén.

# 10 Použití aplikace

Základní použití a běh aplikace je znázorněn na obr. 20 Aktivní diagram. Program vygeneruje defaultní počet vstupů. Pokud uživatel potřebuje zadání upravit, pak si jej musí přenastavit. Je možné také nastavit, kolik se má nalézt řešení, než se ukončí vyhledávající algoritmus. Po úpravě nastavení uživatel vyplní potřebné vstupy, jejich pořadí a omezující funkce. Další možností je načtení ukázkových příkladů, kde jsou již vyplněné domény a omezení, je ponechán počet nalezených řešení do ukončení prohledávání. Dalším krokem je vybrání algoritmu k prohledávání. Posledním krokem je vykreslení grafu. Graf se může vykreslit po jednom nebo deseti krocích dopředu nebo zpátky, další možnost je nechat si vykreslit celý graf pomocí jednoho tlačítka. Následně si uživatel může graf přizpůsobit pro jeho lepší přehlednost pomocí přiblížení, oddálení nebo přesouvání jednotlivých uzlů.




*Obr. 20 Aktivitní diagram od nastavení vstupů až po vykreslení grafu*

# 11 Nasazení

Aplikace je napsána v programovacím jazyku TypeScript a postavená na frameworku Angular5. Zdrojové kódy jsou přiloženy k diplomové práci v souboru source.zip. Tyto kódy se musí přeložit do verze připravené k distribuci na aplikační server. K překladu je nutné mít instalován Angular CLI[7].

1. Rozbalit komprimovaný soubor source.zip do vlastního adresáře.
2. Spustit překlad zdrojových kódu příkazem
   ng build –prod –build-optimizer

   Příznak –prod znamená, že bude vytvořena produkční distribuce, do které nebudou zahrnuty nástroje pro lazení a odstraňování chyb. Příznak –build-optimizer dále vylepšuje překlad pro optimální výkon. [https://angular.io/guide/deployment#build-with---prod]
3. Produkční distribuce je uložena do adresáře dist.

K diplomové práci je přiložena produkční distribuce vyvinuté aplikace v souboru dist.zip. Tu je potřeba nasadit na aplikační server a zpřístupnit ji uživatelům. Aplikace obsahuje serverový skript v PHP, proto je nutné, aby server tento programovací jazyk podporoval. Doporučeným aplikačním serverem je Apache[8].

1. Obsah adresáře dist z předchozího postupu nebo obsah souboru dist.zip zkopírujeme do složky na serveru.
2. Pokud aplikaci nakopírujeme mimo serverový kořenový adresář, je nutné nastavit <base href> v souboru index.html pro podsložku, kde bude aplikace nasazena. Angular tento tag využívá pro směrování požadavků na server. Bez jeho správného nastavení nebude aplikace pracovat správně! Pokud bude dostupná v podsložce tutorial, nastavíme tag na <base href="/tutorial/">
3. Nakonfigurovat aplikační server, aby přesměrovával požadavky na chybějící soubory na index.html
4. Spustit aplikační server

# 12 Testování

Pro testování byly vytvořeny automatické testy. Tyto testy poskytují možnost opětovného spouštění velkých množství testů. Tyto testy tak neustále ověřují veškeré předem nastavené funkcionality a požadavky[9].

V případě, že bude potřeba program jakkoliv upravit, pak se musí také upravit, případně napsat nové testy a také kód programu. Pokud by potom tyto změny v programu vedly k jeho nefunkčnosti, jak v částech, kterou upravujeme, nebo částech již hotových, tak tyto testy nefunkčnosti rychle odhalí. Protože tento přístup vytváří program z malých částí, je mnohem snazší, vypořádat se s různými vnějšími zásahy[9].

Základem je testování aplikační vrstvy pomoci jednotkových testů, které se zaměřují na nejmenší části programu. V tomto programu jsou touto jednotkou implementace jednotlivých variant algoritmu backtrackingu. Vypracoval jsem několik testovacích scénářů, které jsou zaměřeny na obecné principy i na jedinečné vlastnosti různých variant backtrackingu. Každý implementovaný algoritmus v rámci testu vykonává testovací scénář a po jeho vykonání se vyhodnocuje počet nalezených řešení a počet přiřazení, které se pro výpočet použije.

Pro psaní automatických testů byla využita knihovna Jasmine. Angular doporučuje její použití a výrazně rozšiřuje její možnosti. Převážně však pro testování grafického rozhraní. Pro testování servisních tříd a algoritmů stačí základní Jasmine.

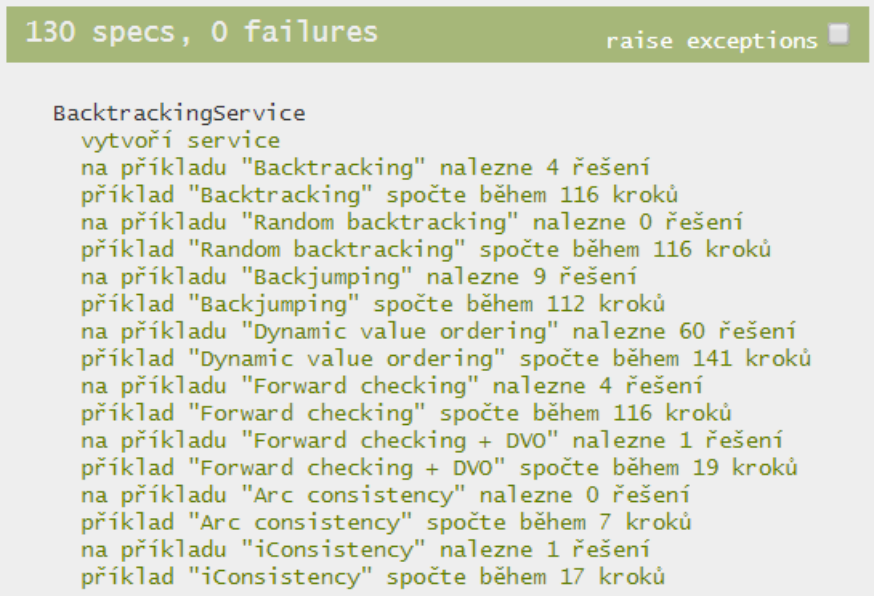
```
describe('BacktrackingService', () => {

  it('na příkladu "Backtracking" nalezne 4 řešení', () => {
    const service = new BacktrackingService();
    const postup = service.run(testUtils.backtrackingExample, 0, 3);
    expect(testUtils.pocetReseni(postup)).toBe(4);
  });
}
```

*Kód 5 - Ukázka testovacího scénáře Backtracking a vyhodnocení počtu řešení.*

Angular CLI umožňuje spouštění všech testů v aplikaci pomocí příkazu: ng test.

Příkaz spustí Karma test runner. Karma vyhledá a spustí všechny testovací scénáře, které se v této aplikaci nacházejí. Tyto scénáře musí být uloženy v souborech s příponou *.spec.ts. Výsledky testů jsou potom vypsány do příkazové řádky a po ukončení testování je zobrazeno závěrečné shrnutí na webové stránce.




*Obr 21 - Ukázka výsledku testování algoritmu Backtracking*

# 13 Závěr

Tato práce je zaměřena na využití ve výuce a proto byla věnována větší pozornost při implementování jednotlivých algoritmů na získání postupu a informací co se v každém kroku děje než efektivitě vyhledání řešení. Pro práci je ideální počet do desíti proměnných, avšak počet proměnných není omezen a aplikace se tedy dá použít i na vyhledání řešení složitějších problémů.

Uživatelské rozhraní aplikace jsem se snažil udělat přehledné prostřednictvím zjednodušení spočívající v zobrazení minima prvků.

Vytvořená aplikace umožňuje uživatelům řešit úlohy v oblasti constraint processing (úlohy s omezením), a to buď zadáním vstupních dat, nebo vypočtením ukázkových příkladů. Při řešení mají uživatelé možnost vybrat si ze čtyř algoritmů, a to základní backtracking, dalším je představitel zpětných algoritmů (look-back) backjumping a představitelé dopředných algoritmů (look-ahead) dopředná kontrola a hranová konzistentnost (arc-consistency). Po zadání hodnot a provedeném výpočtu si uživatelé mohou nechat postup vykreslit jako graf ve formě stromové struktury.

Pro vytvoření aplikace byl použit jazyk JavaScript, a to především kvůli jeho rychlosti a snadné nasaditelnosti. Jelikož byla pro práci využita také technologie PHP, musí být aplikace nasazena na serveru podporující PHP.

Aplikace bude sloužit především jako podpůrný prostředek ve výuce zabývající se oblastí constraint processing, především v předmětu Automatizované řešení úloh s omezením (ARUO).

Aplikace byla vytvořena proto, že se při vyhledávání stávajícího softwaru nenalezla žádná aplikace, která by měla na výběr zvolení algoritmů, kterým se má provést výpočet. Navíc tyto aplikace slouží pouze ke zjištění řešení, ale nemají zobrazen postup, jak algoritmus probíhal, nebo chybí možnost si tento postup vykreslit pomocí grafu.

Vytvořením funkční aplikace pro výpočty z oblasti constraint processing bylo dosaženo stanoveného cíle této práce – vytvořit server pro podporu výuky ARUO.

Tato aplikace umožnuje další rozšíření. Mezi tato rozšíření patří implementace dalších algoritmů.

# 14 Literatura

[1] Rina Dechter, *Constraint Processing*, Morgan Kaufmann Publishers 2003, ISBN: 978-1-55860-890-0

[2] Martin Kot [online, citováno 22. dubna 2016], Slidy, http://www.cs.vsb.cz/kot/download/aruo2013/aruo.pdf

[3] ROSSI, Francesca, VAN BEEK, Peter a Toby WALSH, ed. Handbook of constraint programming. Boston, MA: Elsevier, 2006. Foundations of artificial intelligence. ISBN 0-444-52726-5.

[4] Patrick Prosser, Hybrid Algorithms for the Constraint Satisfaction Problem. Computational Intelligence 9: 268-299, 1993

[5] GoJS Diagrams for JavaScript and HTML, by Northwoods Software. *Document Moved* [online]. Dostupné z: https://gojs.net/latest/index.html

[6] *JSON* [online]. Dostupné z: http://www.json.org/json-cz.html

[7] Angular Docs. *Angular Docs* [online]. Dostupné z: https://angular.io/tutorial/toh-pt0

[8] Welcome! - The Apache HTTP Server Project. *Welcome! - The Apache HTTP Server Project* [online]. Copyright © 1997 [cit. 30.04.2018]. Dostupné z: https://httpd.apache.org/

[9] Programování řízené testy – Wikipedie. [online]. Dostupné z: https://cs.wikipedia.org/wiki/Programov%C3%A1n%C3%AD_%C5%99%C3%ADzen%C3%A9_testy

# 15 Přílohy

Obsah CD

- Samostatná práce ve formátu pdf
- Zdrojové kódy ve formě html a JavaScript souborů
- Manuál pro práci s aplikací